

Projet Machine Learning

Apprentissage avec des Benchmarks &
Détection de Fraudes Bancaires

Youssra Bouhanna & Melissa Bouloufa

M2 Informatique BI&A

Février 2026

Table des matières

I	Apprentissage avec des Benchmarks	3
1	Introduction	3
2	Méthodologie	4
2.1	Notations	4
2.2	Mesures de performance	4
2.3	Algorithmes et variantes	5
2.3.1	KNN (k-plus proches voisins)	5
2.3.2	Modèles linéaires	5
2.3.3	Modèles non linéaires	6
2.4	Pipelines d'entraînement	7
3	Expériences	8
3.1	Protocole expérimental	8
3.2	Résultats	8
3.2.1	KNN	8
3.2.2	Modèles linéaires	10
3.2.3	Modèles non linéaires	12
3.3	Comparaison globale	14
3.4	Temps d'apprentissage	14
4	Conclusion	15
II	Détection de Fraudes Bancaires	17
5	Introduction	17
5.1	Contexte	17
5.2	Objectifs du projet	17
5.3	Jeu de données et caractéristiques	17
5.4	Déséquilibre des classes	18
6	Méthodologie	18
6.1	Notations	18
6.2	Mesures de performance	19
6.3	Familles de modèles	20
6.4	Techniques de traitement du déséquilibre	21
6.5	Approches spécifiques implémentées	22
7	Expériences	23
7.1	Protocole expérimental	23
7.2	Choix des variables	23
7.3	Algorithmes testés	23
7.3.1	KNN	24
7.3.2	Régression Logistique	25
7.3.3	Random Forest	27

7.3.4	K-means	31
7.3.5	XGBoost	32
7.4	Optimisation des hyperparamètres	35
8	Résultats	36
8.1	Présentation des résultats	36
8.2	Analyse et interprétation	37
9	Conclusion et perspectives	38
9.1	Synthèse des résultats	38
9.2	Limites du travail	38
9.3	Perspectives	39
A	Annexe : Meilleurs modèles par dataset	40

Première partie

Apprentissage avec des Benchmarks

1 Introduction

Ce projet porte sur la classification binaire appliquée à un ensemble de jeux de données classiques de la communauté Machine Learning. L'objectif est de comparer plusieurs familles d'algorithmes (non paramétriques, linéaires, non linéaires) et de proposer des variantes pour améliorer les performances, notamment sur les jeux de données déséquilibrés.

On dispose de 28 jeux de données de tailles et caractéristiques variées. Le tableau 1 résume les propriétés principales de quelques-uns d'entre eux.

TABLE 1 – Caractéristiques des jeux de données (extrait).

Dataset	n	d	Ratio positif	Déséquilibré
abalone8	4 177	10	13.6%	oui
abalone17	4 177	10	1.4%	oui
abalone20	4 177	10	0.6%	oui
australian	690	14	44.5%	non
balance	625	4	46.1%	non
bankmarketing	45 211	51	11.7%	oui
german	1 000	24	30.0%	non
hayes	132	4	22.7%	non
iono	351	34	35.9%	non
libras	360	90	6.7%	oui
pageblocks	5 473	10	10.2%	oui
pima	768	8	34.9%	non
segmentation	2 310	19	14.3%	oui
vehicle	846	18	23.5%	non
wine	178	13	33.1%	non
wine4	1 599	11	3.3%	oui

On considère un jeu de données comme déséquilibré lorsque la classe minoritaire représente moins de 20% des exemples, ce qui correspond à un ratio d'au moins 80/20 généralement vu comme un déséquilibre significatif. Parmi les 28 datasets, une bonne partie est déséquilibrée, ce qui justifie l'utilisation de techniques d'échantillonnage et de pondération des classes.

Après préparation (fonction `data_recovery`), aucun dataset ne contient de valeurs manquantes.

2 Méthodologie

2.1 Notations

On note :

- $X \in \mathbb{R}^{n \times d}$ la matrice des features, avec n le nombre d'exemples et d la dimension.
- $y \in \{0, 1\}^n$ le vecteur des labels binaires.
- X_{train}, X_{test} les sous-ensembles d'entraînement et de test.
- \hat{y} les prédictions du modèle sur X_{test} .

2.2 Mesures de performance

Nous explorons plusieurs métriques pour évaluer les modèles.

- **Accuracy**

Proportion de prédictions correctes. Elle est simple à interpréter, mais peut être trompeuse sur les jeux de données déséquilibrés : un modèle qui prédit toujours la classe majoritaire peut obtenir une accuracy élevée sans réellement apprendre.

- **Précision**

Proportion de prédictions positives qui sont correctes, définie par :

$$\text{Precision} = \frac{TP}{TP + FP}$$

où TP désigne les vrais positifs et FP les faux positifs.

- **Rappel**

Proportion de positifs réellement détectés par le modèle, définie par :

$$\text{Recall} = \frac{TP}{TP + FN}$$

où FN désigne les faux négatifs.

- **F1-score**

Moyenne harmonique de la précision et du rappel, définie par :

$$F_1 = \frac{2TP}{2TP + FN + FP}$$

Le F1-score est notre métrique principale de comparaison, car il combine précision et rappel et pénalise simultanément les faux positifs et les faux négatifs. Il est donc particulièrement adapté aux situations de classes déséquilibrées, où une accuracy élevée peut masquer des performances très faibles sur la classe minoritaire.

- **AUC ROC**

Aire sous la courbe ROC, qui mesure la capacité du modèle à distinguer les deux classes indépendamment du seuil de décision.

Nous mesurons également le temps d'apprentissage de chaque modèle afin de comparer leurs performances en termes de rapidité.

2.3 Algorithmes et variantes

2.3.1 KNN (k-plus proches voisins)

Le KNN classe un exemple x en regardant ses k voisins les plus proches et en prenant un vote majoritaire :

$$\hat{y}(x) = \arg \max_{c \in \{0,1\}} \sum_{i \in \mathcal{N}_k(x)} \mathbb{1}[y_i = c]$$

où $\mathcal{N}_k(x)$ désigne les k voisins les plus proches.

Nous avons testé 5 variantes :

1. **KNN de base** : $k = 5$, distance euclidienne, poids uniformes. Version simple sans pré-traitement.
2. **KNN pondéré** : $k = 5$, mais les voisins proches comptent plus (`weights="distance"`).
3. **KNN + normalisation** : ajout d'un `StandardScaler` avant le KNN. Comme le KNN se base sur les distances, les features avec de grandes valeurs dominent le calcul si on ne normalise pas.
4. **KNN + GridSearch** : recherche du meilleur $k \in \{1, 3, 5, 7, 11\}$ via validation croisée (5-fold) avec scoring F1, après normalisation.
5. **KNN + SMOTE** : combine SMOTE avec normalisation et KNN. SMOTE génère des exemples synthétiques de la classe minoritaire. Appliqué uniquement sur les jeux déséquilibrés.

2.3.2 Modèles linéaires

Nous testons deux familles de modèles linéaires, la régression logistique et le SVM linéaire, chacune déclinée en trois variantes.

Régression logistique

La régression logistique modélise la probabilité qu'un exemple appartienne à la classe positive via une fonction sigmoïde :

$$P(y = 1 | x) = \sigma(w^T x + b) = \frac{1}{1 + e^{-(w^T x + b)}}.$$

Le modèle est entraîné en minimisant la log-vraisemblance négative (binary cross-entropy) avec régularisation L_2 :

$$\mathcal{L}(w) = -\frac{1}{n} \sum_{i=1}^n [y_i \log(\hat{p}_i) + (1 - y_i) \log(1 - \hat{p}_i)] + \frac{1}{2C} \|w\|^2,$$

où $C > 0$ contrôle le compromis entre ajustement aux données et régularisation.

Nous considérons trois variantes, toutes précédées d'une standardisation des variables :

- **LogReg base** : pénalité L_2 , $C = 1.0$, solveur `lbfgs`. Cette variante sert de référence « classique ».
- **LogReg balanced** : même configuration, avec `class_weight="balanced"`, qui ajuste automatiquement les poids de classe

$$w_j = \frac{n_{\text{samples}}}{n_{\text{classes}} \cdot n_j},$$

afin de pénaliser davantage les erreurs sur la classe minoritaire dans les jeux déséquilibrés.

- **LogReg L1** : pénalité L_1 avec $C = 1.0$ et solveur `liblinear`, ce qui induit une solution parcimonieuse (certains coefficients sont exactement nuls) et joue le rôle d'une sélection de variables implicite.

SVM linéaire

Le SVM linéaire cherche l'hyperplan $w^T x + b = 0$ qui sépare au mieux les deux classes, en maximisant la marge tout en contrôlant les erreurs via le paramètre C :

$$\min_{w,b} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^n \ell(y_i, w^T x_i + b),$$

où ℓ désigne une variante de la hinge loss.

Nous utilisons également trois variantes, elles aussi précédées d'une standardisation :

- **LinearSVC base** : perte `squared_hinge` (par défaut) et $C = 1.0$, qui correspond à une version quadratique de la hinge loss.
- **LinearSVC balanced** : même paramétrage, avec `class_weight="balanced"` pour gérer les jeux déséquilibrés en pondérant davantage la classe minoritaire, selon la même formule que pour la régression logistique.
- **LinearSVC hinge** : même configuration mais avec `loss="hinge"`, qui utilise la hinge loss standard, ce qui modifie légèrement la manière dont les erreurs proches de la marge sont pénalisées.

Pour les variantes marquées `balanced`, l'approche est dite *cost-sensitive* : au lieu de rééchantillonner les données, on modifie la fonction de coût pour rendre les erreurs sur la classe minoritaire plus coûteuses et inciter le modèle à m

2.3.3 Modèles non linéaires

Quatre algorithmes testés :

1. **Arbre de décision** : partitionne l'espace par seuils successifs. Simple mais sujette au surapprentissage.
2. **Random Forest** : ensemble de 100 arbres sur sous-échantillons bootstrap. Le vote majoritaire réduit la variance.
3. **AdaBoost** : boosting séquentiel donnant plus de poids aux exemples mal classés. 100 estimateurs, `learning_rate=1.0`.
4. **Gradient Boosting** : optimise une fonction de perte par descente de gradient. 100 estimateurs, `learning_rate=0.1, max_depth=3`.

Pour ces quatre algorithmes, une variante avec **ADASYN** (Adaptive Synthetic Sampling) est proposée sur les datasets déséquilibrés. ADASYN fonctionne comme SMOTE mais génère plus d'exemples synthétiques dans les zones où la classe minoritaire est difficile à apprendre. Sur les jeux équilibrés, on ne l'applique pas car cela peut dégrader les résultats.

2.4 Pipelines d'entraînement

Les variantes utilisant du rééchantillonnage sont implémentées via des pipelines `imblearn`. Le pseudo-code ci-dessous illustre les principaux pipelines.

Algorithm 1 Pipeline SMOTE + KNN (datasets déséquilibrés)

-
- 1: Split stratifié : $(X_{\text{train}}, X_{\text{test}}, y_{\text{train}}, y_{\text{test}}) \leftarrow \text{train_test_split}(X, y, \text{test_size} = 0.3, \text{stratify} = y)$
 - 2: Sur-échantillonnage : $(X_{\text{res}}, y_{\text{res}}) \leftarrow \text{SMOTE.fit_resample}(X_{\text{train}}, y_{\text{train}})$
 - 3: Normalisation : $X_{\text{res}, \text{scaled}} \leftarrow \text{StandardScaler.fit_transform}(X_{\text{res}})$
 - 4: Entraînement : $\text{KNN.fit}(X_{\text{res}, \text{scaled}}, y_{\text{res}})$
 - 5: Prédiction : $\hat{y} \leftarrow \text{KNN.predict}(\text{StandardScaler.transform}(X_{\text{test}}))$
-

Algorithm 2 Pipeline ADASYN + Gradient Boosting

-
- 1: Split stratifié : $(X_{\text{train}}, X_{\text{test}}, y_{\text{train}}, y_{\text{test}}) \leftarrow \text{train_test_split}(X, y, \text{test_size} = 0.3, \text{stratify} = y)$
 - 2: Sur-échantillonnage adaptatif : $(X_{\text{res}}, y_{\text{res}}) \leftarrow \text{ADASYN.fit_resample}(X_{\text{train}}, y_{\text{train}})$
 - 3: Entraînement : $\text{GradientBoosting.fit}(X_{\text{res}}, y_{\text{res}})$
 - 4: Prédiction : $\hat{y} \leftarrow \text{GradientBoosting.predict}(X_{\text{test}})$
-

Algorithm 3 Pipeline KNN avec `GridSearchCV`

-
- 1: Définir un pipeline : `StandardScaler` → `KNeighborsClassifier`
 - 2: Définir la grille de paramètres : $k \in \{1, 3, 5, 7, 11\}$
 - 3: Recherche du meilleur modèle : `GridSearchCV(pipeline, cv=5, scoring="f1")`
 - 4: Sélection du meilleur k par validation croisée
 - 5: Entraînement du modèle final sur $(X_{\text{train}}, y_{\text{train}})$
-

3 Expériences

3.1 Protocole expérimental

- **Données** : 28 datasets de classification binaire, de 132 à 45 211 exemples, 4 à 90 features.
- **Découpe** : split train/test stratifié 70/30 (`test_size=0.3, random_state=0`). La stratification conserve le ratio des classes.
- **Hyper-paramètres** : fixés à des valeurs par défaut pour la plupart des modèles. Seul le KNN avec `GridSearchCV` fait une recherche sur $k \in \{1, 3, 5, 7, 11\}$ (5-fold, scoring F1).
- **Itérations** : chaque modèle est entraîné et évalué sur le même split train/test lors de 5 itérations indépendantes ; les tableaux rapportent la moyenne et l'écart-type des F_1 -scores.
- **Datasets déséquilibrés** : SMOTE, ADASYN et `class_weight="balanced"` ne sont appliqués que sur les jeux avec classe minoritaire < 20 %.

Toutes les évaluations utilisent la même fonction `evaluate_model`.

3.2 Résultats

3.2.1 KNN

Le tableau 2 présente les F_1 -scores des variantes KNN. Le tiret (-) indique que la variante n'est pas appliquée sur ce dataset.

Dataset	Imbalanced	knn_base	knn_distance	knn_scaled	knn_cv	knn_smote
abalone8	True	0.191 ± 0.029	0.184 ± 0.027	0.196 ± 0.023	0.241 ± 0.024	0.324 ± 0.023
abalone17	True	0.000 ± 0.000	0.021 ± 0.047	0.000 ± 0.000	0.060 ± 0.043	0.071 ± 0.031
abalone20	True	0.000 ± 0.000	0.000 ± 0.000	0.000 ± 0.000	0.022 ± 0.050	0.068 ± 0.025
australian	False	0.568 ± 0.029	0.578 ± 0.016	0.815 ± 0.027	0.819 ± 0.023	—
balance	False	0.899 ± 0.019	0.899 ± 0.019	0.891 ± 0.025	0.938 ± 0.034	—
bankmarketing	True	0.348 ± 0.009	0.354 ± 0.010	0.407 ± 0.012	0.410 ± 0.012	0.464 ± 0.007
libras	True	0.705 ± 0.096	0.732 ± 0.101	0.692 ± 0.140	0.811 ± 0.093	0.758 ± 0.165
segmentation	True	0.828 ± 0.024	0.878 ± 0.035	0.838 ± 0.027	0.898 ± 0.027	0.849 ± 0.041
vehicle	False	0.843 ± 0.010	0.847 ± 0.015	0.848 ± 0.017	0.857 ± 0.028	—
wine	False	0.849 ± 0.035	0.862 ± 0.034	0.953 ± 0.021	0.963 ± 0.023	—
wine4	True	0.000 ± 0.000	0.000 ± 0.000	0.000 ± 0.000	0.049 ± 0.028	0.103 ± 0.041

TABLE 2 – Extrait des performances F_1 -moyennes (± écart-type) des variantes KNN.

Observations :

- Le KNN de base échoue sur certains jeux très déséquilibrés ($F_1 \approx 0$ sur *abalone17*, *abalone20*, *wine4*), ce qui reste logique : quand la classe positive est très rare, ses k voisins sont presque toujours de la classe majoritaire.
- La pondération par distance (`knn_distance`) apporte globalement peu de gains par rapport à `knn_base` (par exemple *australian* 0.568 → 0.578, *german* 0.400 → 0.404), avec quelques améliorations ponctuelles mais aucune tendance massive.
- La normalisation (`knn_scaled`) aide sur plusieurs jeux où les features n'ont pas la même échelle, comme *australian* (0.568 → 0.815) ou *spambase* (0.740 → 0.881), alors que sur d'autres jeux (*balance*, *segmentation*) l'effet reste limité.

- L'optimisation de k par validation croisée (`knn_cv`) améliore systématiquement ou presque les performances par rapport à `knn_base` (par exemple *libras* 0.705 → 0.811, *wdbc* 0.901 → 0.947), ce qui montre que choisir le bon k est crucial.
- SMOTE (`knn_smote`) améliore nettement le F1 sur plusieurs jeux déséquilibrés, par exemple *abalone8* (0.191 → 0.324), *abalone17* (0.000 → 0.071) ou *wine4* (0.000 → 0.103), mais peut aussi légèrement dégrader les scores sur certains jeux comme *yeast3* (0.780 → 0.685), ce qui illustre le fait que le sur-échantillonnage n'est pas toujours bénéfique.

La figure 1 permet de visualiser plus clairement, pour chaque dataset, la variante de KNN la plus performante.

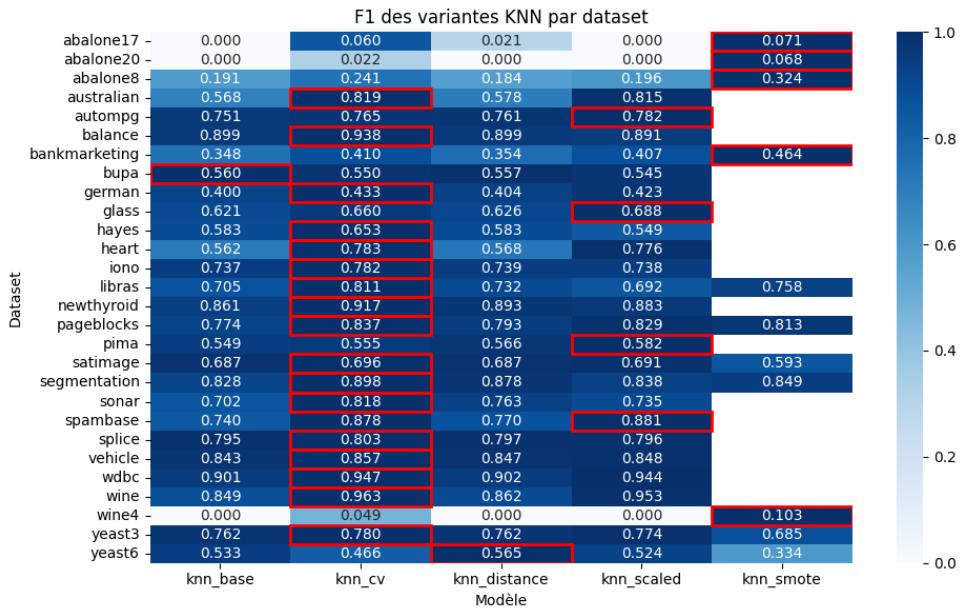


FIGURE 1 – Carte de chaleur des F1-scores des variantes KNN par dataset.

Temps d'apprentissage : Au-delà des scores de F1, nous comparons aussi le temps d'apprentissage moyen de chaque variante de KNN. La figure 2 montre que la variante `knn_cv`, qui effectue une recherche de k par validation croisée, est de loin la plus coûteuse en temps de calcul, alors que les autres variantes restent très rapides. La version avec SMOTE (`knn_smote`) introduit un surcoût modéré lié au sur-échantillonnage préalable, mais reste nettement plus abordable que `knn_cv`. Cela illustre le compromis classique entre performance et coût de calcul.

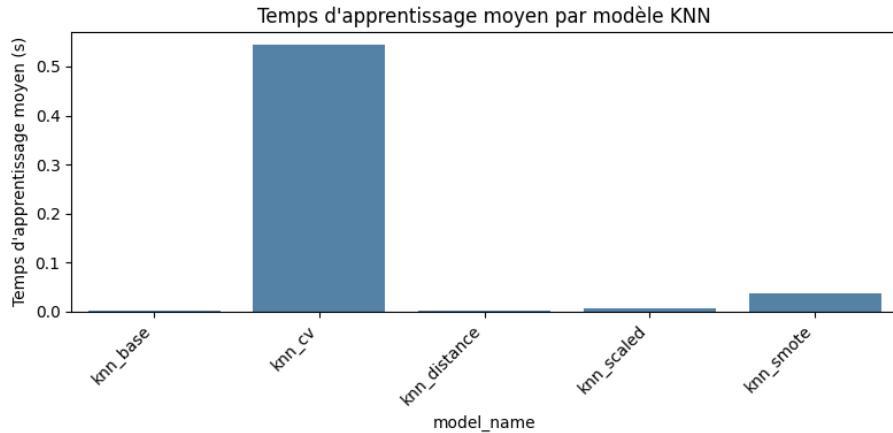


FIGURE 2 – Temps d'apprentissage moyen par variante de KNN (en secondes).

3.2.2 Modèles linéaires

Le tableau 3 présente un extrait des F_1 -scores moyens (\pm écart-type) pour les modèles linéaires. Le tiret (–) indique que la variante n'est pas appliquée sur ce dataset.

Dataset	Imbalanced	logreg_base	logreg_balanced	logreg_l1	svm_lin_base	svm_lin_hinge	svm_lin_balanced
abalone8	True	0.007 \pm 0.010	0.369 \pm 0.013	0.007 \pm 0.010	0.005 \pm 0.006	0.000 \pm 0.000	0.365 \pm 0.011
abalone17	True	0.000 \pm 0.000	0.097 \pm 0.008	0.000 \pm 0.000	0.000 \pm 0.000	0.000 \pm 0.000	0.099 \pm 0.011
abalone20	True	0.000 \pm 0.000	0.064 \pm 0.011	0.000 \pm 0.000	0.000 \pm 0.000	0.000 \pm 0.000	0.070 \pm 0.007
australian	False	0.839 \pm 0.022	–	0.846 \pm 0.017	0.847 \pm 0.007	0.842 \pm 0.011	–
balance	False	0.949 \pm 0.017	–	0.949 \pm 0.017	0.949 \pm 0.017	0.949 \pm 0.017	–
bankmarketing	True	0.451 \pm 0.011	0.551 \pm 0.010	0.452 \pm 0.011	0.416 \pm 0.014	0.287 \pm 0.010	0.556 \pm 0.010
segmentation	True	0.622 \pm 0.022	0.628 \pm 0.011	0.622 \pm 0.021	0.634 \pm 0.021	0.644 \pm 0.030	0.620 \pm 0.013
wine	False	0.989 \pm 0.015	–	0.983 \pm 0.015	0.994 \pm 0.013	0.989 \pm 0.024	–
wine4	True	0.000 \pm 0.000	0.133 \pm 0.024	0.000 \pm 0.000	0.000 \pm 0.000	0.000 \pm 0.000	0.131 \pm 0.023

TABLE 3 – Extrait des performances F_1 moyennes (\pm écart-type) des modèles linéaires.

Observations :

- Les variantes linéaires de base donnent toujours un F1 quasi nul sur les jeux très déséquilibrés (*abalone17*, *abalone20*, *wine4*), quel que soit le modèle (régression logistique ou SVM), ce qui montre qu'elles prédisent presque uniquement la classe majoritaire.
- L'option `class_weight="balanced"` améliore nettement ces jeux difficiles : par exemple sur *abalone8*, la régression logistique passe de environ 0.01 à 0.37, et le SVM linéaire de environ 0.01 à 0.37–0.38, ce qui montre que la pondération de la classe minoritaire permet enfin de la détecter.
- Sur les jeux globalement équilibrés, les différentes variantes de LogReg et SVM linéaire obtiennent des F1 très proches et élevés (par exemple *balance*, *australian*, *wine*), ce qui confirme que les modèles linéaires capturent bien les frontières de décision lorsque la séparation est à peu près linéaire.
- Les variantes avec pénalité L_1 et les différentes pertes du SVM (hinge, squared hinge) donnent des performances proches des versions L_2 , avec des écarts limités selon les jeux, ce qui suggère que le choix de la pondération des classes a plus d'impact que le choix fin de la variante linéaire.

La figure 3 permet de visualiser plus clairement, pour chaque dataset, la variante linéaire la plus performante.

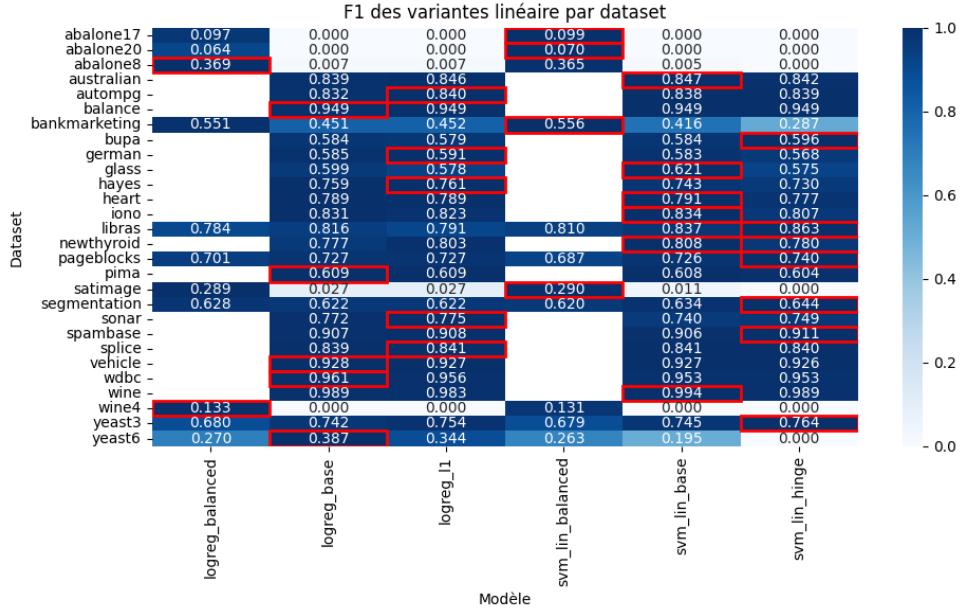


FIGURE 3 – Carte de chaleur des F1-scores des variantes linéaires par dataset.

Temps d'apprentissage : La figure 4 montre que les temps d'apprentissage des modèles linéaires restent très faibles, mais avec quelques différences notables. La régression logistique avec pénalité L_1 (logreg_11) est nettement plus coûteuse que la version L_2 (logreg_base et logreg_balanced), en raison de l'optimisation plus complexe imposée par la régularisation L_1 . De même, la variante svm_lin_hinge est la plus lente parmi les SVM linéaires, tandis que svm_lin_base et svm_lin_balanced restent très rapides. Globalement, même les variantes les plus lourdes restent bien moins coûteuses que les modèles KNN avec recherche de k par validation croisée.

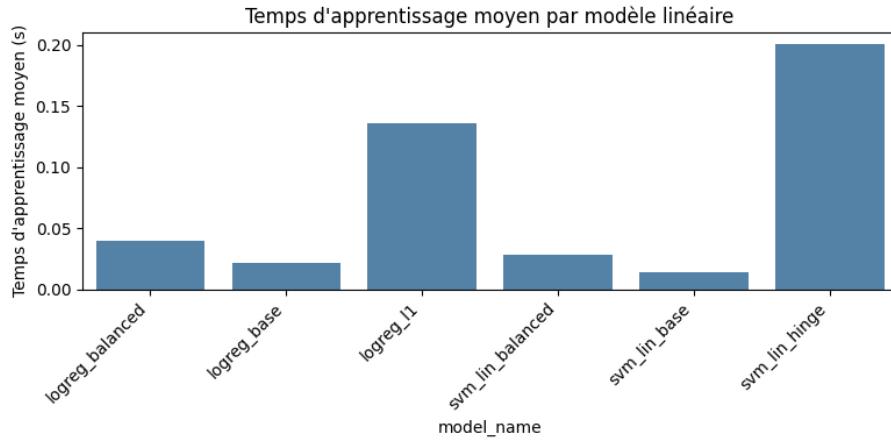


FIGURE 4 – Temps d'apprentissage moyen par modèle linéaire (en secondes).

3.2.3 Modèles non linéaires

Le tableau 4 présente un extrait des F_1 -scores moyens (\pm écart-type) pour les modèles non linéaires. Le tiret (-) indique que la variante n'est pas appliquée sur ce dataset.

Dataset	Imbalanced	tree	rf	ada	gb	tree_adas.	rf_adas.	gb_adas.
abalone8	True	0.239 \pm 0.024	0.130 \pm 0.023	0.000 \pm 0.000	0.111 \pm 0.023	0.269 \pm 0.014	0.298 \pm 0.013	0.385 \pm 0.018
abalone17	True	0.041 \pm 0.065	0.000 \pm 0.000	0.000 \pm 0.000	0.000 \pm 0.000	0.052 \pm 0.019	0.062 \pm 0.030	0.097 \pm 0.024
abalone20	True	0.000 \pm 0.000	0.000 \pm 0.000	0.000 \pm 0.000	0.000 \pm 0.000	0.051 \pm 0.064	0.115 \pm 0.081	0.077 \pm 0.048
australian	False	0.772 \pm 0.022	0.848 \pm 0.023	0.832 \pm 0.015	0.839 \pm 0.025	—	—	—
balance	False	0.889 \pm 0.022	0.903 \pm 0.013	0.982 \pm 0.012	0.928 \pm 0.015	—	—	—
bankmarketing	True	0.485 \pm 0.005	0.493 \pm 0.010	0.448 \pm 0.012	0.505 \pm 0.012	0.483 \pm 0.010	0.500 \pm 0.009	0.567 \pm 0.015
glass	False	0.681 \pm 0.059	0.766 \pm 0.059	0.717 \pm 0.027	0.785 \pm 0.054	—	—	—
libras	True	0.633 \pm 0.192	0.659 \pm 0.178	0.796 \pm 0.163	0.757 \pm 0.160	0.596 \pm 0.155	0.817 \pm 0.088	0.751 \pm 0.047
pageblocks	True	0.836 \pm 0.010	0.889 \pm 0.011	0.812 \pm 0.017	0.871 \pm 0.013	0.825 \pm 0.014	0.870 \pm 0.017	0.839 \pm 0.008
segmentation	True	0.861 \pm 0.022	0.920 \pm 0.009	0.781 \pm 0.023	0.906 \pm 0.012	0.886 \pm 0.025	0.912 \pm 0.014	0.890 \pm 0.015
spambase	False	0.884 \pm 0.007	0.938 \pm 0.005	0.920 \pm 0.008	0.931 \pm 0.009	—	—	—
vehicle	False	0.840 \pm 0.007	0.934 \pm 0.036	0.924 \pm 0.024	0.923 \pm 0.022	—	—	—
wine	False	0.887 \pm 0.024	0.983 \pm 0.015	0.915 \pm 0.043	0.907 \pm 0.030	—	—	—
wine4	True	0.062 \pm 0.008	0.000 \pm 0.000	0.000 \pm 0.000	0.033 \pm 0.075	0.134 \pm 0.057	0.140 \pm 0.056	0.103 \pm 0.056

TABLE 4 – Extrait des performances F_1 moyennes (\pm écart-type) des modèles non linéaires.

Observations :

- Le Random Forest de base reste l'un des meilleurs modèles non linéaires sur la majorité des jeux équilibrés (*australian*, *spambase*, *vehicle*, *wine*), avec des F1 souvent supérieurs à 0.9, ce qui confirme l'efficacité de l'agrégation d'arbres pour réduire la variance.
- Le Gradient Boosting est très compétitif et parfois légèrement meilleur que le Random Forest sur certains jeux (*glass*, *segmentation*, *pageblocks*), mais les écarts restent généralement modestes et dépendent du dataset.
- AdaBoost fonctionne bien sur plusieurs jeux équilibrés (par exemple *balance*, *hayes*, *spambase*), mais reste fragile sur les jeux très déséquilibrés comme *abalone8*, *abalone17*, *abalone20* ou *wine4*, où le F1 reste proche de 0 sans sur-échantillonnage, signe qu'il se concentre surtout sur la classe majoritaire.
- L'arbre de décision seul est systématiquement en dessous des méthodes d'ensemble (Random Forest, Gradient Boosting, AdaBoost), ce qui est attendu puisque l'agrégation permet de lisser le sur-apprentissage d'un arbre unique.
- L'utilisation d'ADASYN améliore nettement les performances sur plusieurs jeux déséquilibrés, en particulier pour Gradient Boosting et Random Forest (par exemple *abalone8*, *abalone17*, *abalone20*, *wine4*), mais apporte des gains plus limités, voire des légères dégradations, sur certains jeux déjà bien modélisés (*pageblocks*, *yeast6*), ce qui illustre que le sur-échantillonnage peut introduire du bruit quand le modèle de base est déjà performant.

La figure 5 permet de visualiser plus clairement, pour chaque dataset, la variante linéaire la plus performante.

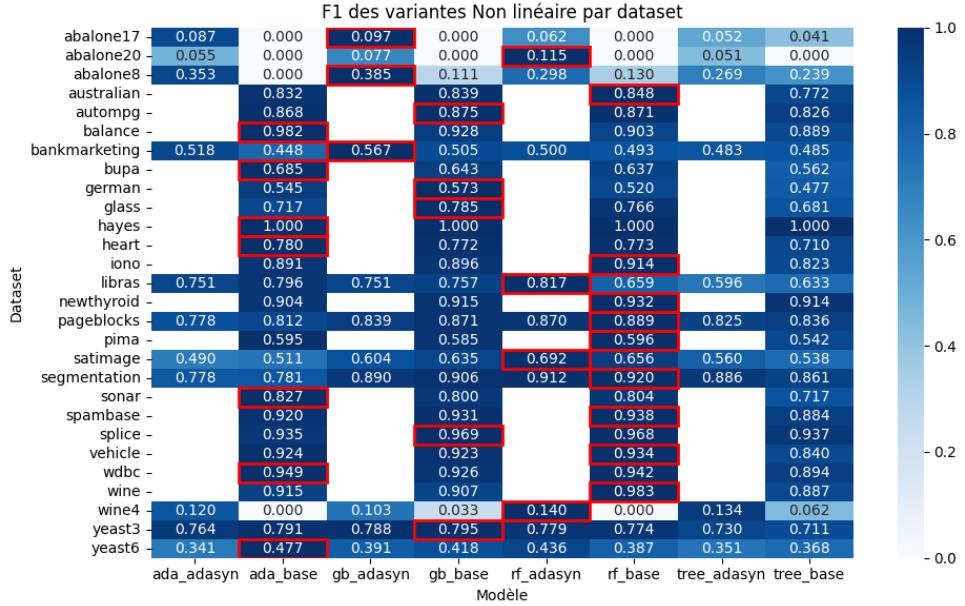


FIGURE 5 – Carte de chaleur des F1-scores des variantes non linéaires par dataset.

Temps d'apprentissage : La figure 6 compare le temps d'apprentissage moyen des modèles non linéaires. Les arbres simples (tree_base) et les Random Forest (rf_base, rf_adasyn) restent relativement rapides à entraîner, malgré le nombre d'arbres. AdaBoost de base (ada_base) est également modéré, mais sa variante sur-échantillonnée (ada_adasyn) devient nettement plus coûteuse. Le Gradient Boosting est le plus lent, surtout avec ADASYN (gb_adasyn), car il combine un grand nombre d'itérations séquentielles avec le sur-échantillonnage, ce qui en fait le modèle non linéaire le plus cher en temps de calcul.

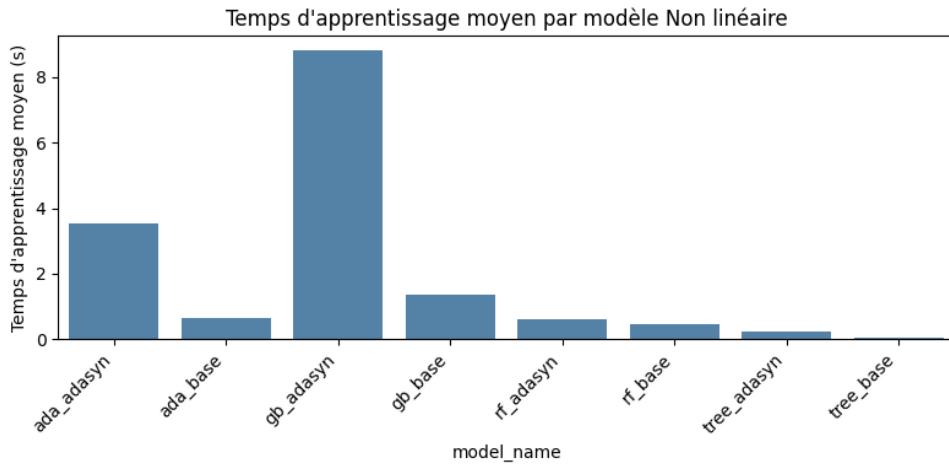


FIGURE 6 – Temps d'apprentissage moyen par modèle non linéaire (en secondes).

3.3 Comparaison globale

L'extrait tableau 5 synthétise, pour chaque dataset, le meilleur modèle de chaque famille et son F_1 , tandis que la version complète est reportée en annexe tableau 8.

Dataset	Imb.	Best KNN	F_1	Best linéaire	F_1	Best non linéaire
abalone17	True	knn_smote	0.071 ± 0.031	svm_lin_balanced	0.099 ± 0.011	gb_adasyn
abalone20	True	knn_smote	0.068 ± 0.025	svm_lin_balanced	0.070 ± 0.007	rf_adasyn
abalone8	True	knn_smote	0.324 ± 0.023	logreg_balanced	0.369 ± 0.013	gb_adasyn
australian	False	knn_cv	0.819 ± 0.023	svm_lin_base	0.847 ± 0.007	rf_base
balance	False	knn_cv	0.938 ± 0.034	logreg_base	0.949 ± 0.017	ada_base
bankmarketing	True	knn_smote	0.464 ± 0.007	svm_lin_balanced	0.556 ± 0.010	gb_adasyn
segmentation	True	knn_cv	0.898 ± 0.027	svm_lin_hinge	0.644 ± 0.030	rf_base
spambase	False	knn_scaled	0.881 ± 0.006	svm_lin_hinge	0.911 ± 0.007	rf_base
vehicle	False	knn_cv	0.857 ± 0.028	logreg_base	0.928 ± 0.026	rf_base
wine	False	knn_cv	0.963 ± 0.023	svm_lin_base	0.994 ± 0.013	rf_base
wine4	True	knn_smote	0.103 ± 0.041	logreg_balanced	0.133 ± 0.024	rf_adasyn
yeast3	True	knn_cv	0.780 ± 0.033	svm_lin_hinge	0.764 ± 0.011	gb_base
yeast6	True	knn_distance	0.565 ± 0.103	logreg_base	0.387 ± 0.177	ada_base

TABLE 5 – Extrait du meilleur modèle par famille et par dataset (F_1 moyen \pm écart-type).

Le tableau 6 résume ensuite les principales tendances observées.

Famille	Points forts	Points faibles
KNN	Simple, pas d'entraînement. knn_cv + SMOTE marche bien sur les jeux déséquilibrés.	Lent en prédiction sur de gros jeux. Sensible aux échelles des features.
Linéaires	Très rapides à entraîner. Bons résultats sur données linéairement séparables.	$F_1 = 0$ sur les jeux très déséquilibrés sans pondération.
Non linéaires	Random Forest et Gradient Boosting donnent les meilleurs F_1 globaux.	Plus longs à entraîner. ADASYN pas toujours utile.

TABLE 6 – Comparaison des trois familles d'algorithme.

Le Gradient Boosting couplé à l'ADASYN sur les jeux déséquilibrés, et le Random Forest sur les jeux équilibrés, représentent les meilleures combinaisons dans nos expériences.

3.4 Temps d'apprentissage

Le temps d'entraînement est mesuré via `perf_counter()` autour de l'appel `model.fit()`. Le tableau 7 illustre les ordres de grandeur des différentes familles observés sur quelques datasets représentatifs.

Dataset	KNN base	LogReg	RF (100 arbres)	GB (100 arbres)
abalone8 ($n=4\,177$)	<0.01	~0.10	~0.30	~0.35
bankmarketing ($n=45\,211$)	~0.02	~0.15	~3.5	~5.0
wine ($n=178$)	<0.01	<0.01	~0.05	~0.04
splice ($n=3\,175$)	<0.01	~0.03	~0.25	~0.50

TABLE 7 – Temps d’apprentissage moyens (en secondes) par famille de modèle.

Observations.

- Le KNN est quasi instantané à l’entraînement (il ne fait que stocker les données), mais son coût se reporte sur la prédiction. Sur les gros jeux comme *bankmarketing*, la prédiction est donc beaucoup plus lente.
- Les modèles linéaires (LogReg, SVM) sont très rapides à entraîner, de l’ordre de quelques centièmes à dixièmes de seconde. C’est leur principal avantage pratique.
- Les méthodes d’ensemble (RF, GB) sont plus coûteuses, proportionnellement au nombre d’arbres et à la taille du dataset. Le Gradient Boosting est séquentiel par nature, tandis que la Random Forest est parallélisable (`n_jobs=-1`).
- Les variantes avec SMOTE/ADASYN ajoutent un surcoût lié à la génération des exemples synthétiques. Le GridSearchCV du KNN multiplie le temps par le nombre de combinaisons \times le nombre de folds (ici $5 \times 5 = 25$ fits).

4 Conclusion

On a testé trois familles d’algorithmes sur 28 datasets de classification binaire : KNN (5 variantes), modèles linéaires (6 variantes) et modèles non linéaires (8 variantes). Le F1-score a été utilisé comme métrique principale car l’accuracy est souvent trompeuse sur les jeux déséquilibrés.

Les principaux enseignements :

- La normalisation des features est indispensable pour les méthodes à base de distance (KNN) et les modèles linéaires.
- Les techniques de rééchantillonnage (SMOTE, ADASYN) et de pondération des classes (`class_weight="balanced"`) améliorent le F1 sur les jeux déséquilibrés, mais peuvent dégrader les résultats sur les jeux déjà équilibrés. Il faut les appliquer au cas par cas.
- Les méthodes d’ensemble (Random Forest, Gradient Boosting) donnent les meilleurs résultats globaux, au prix d’un temps d’entraînement plus élevé.
- Certains jeux restent très difficiles (*abalone20* avec 0.6% de positifs, *wine4* avec 3.3%) : même avec du rééchantillonnage, les F1 restent faibles.

Perspectives : Comme pistes d’amélioration, on pourrait explorer :

- Un tuning plus poussé des hyper-paramètres (GridSearch ou RandomSearch sur C , `learning_rate`, `max_depth`, `n_estimators`) pour chaque dataset.
- Des méthodes de combinaison de modèles (stacking, voting) pour tirer parti des forces de chaque algorithme.

- L'utilisation de Tomek Links ou Edited Nearest Neighbours en under-sampling, non testés ici, qui pourraient nettoyer les frontières de décision mieux que le sur-échantillonnage seul.

Deuxième partie

Détection de Fraudes Bancaires

5 Introduction

5.1 Contexte

La détection de fraude occupe une place centrale dans le secteur bancaire et, plus largement, dans tous les systèmes de paiement électroniques. Les institutions financières doivent protéger leurs clients et limiter les pertes financières tout en maintenant une expérience utilisateur fluide, ce qui rend indispensable l'usage de modèles capables de signaler rapidement les transactions suspectes.

Dans ce contexte, les acteurs font face à une problématique opérationnelle typique : un volume très important de transactions quotidiennes, une proportion de fraudes extrêmement faible et des attaques de plus en plus sophistiquées. Cette combinaison rend impossible un contrôle manuel systématique et impose le recours à des méthodes d'apprentissage automatique pour automatiser la détection, prioriser les alertes et assister les équipes de lutte contre la fraude.

5.2 Objectifs du projet

L'objectif principal de ce projet est de construire et comparer plusieurs modèles de classification capables d'identifier au mieux la classe fraude à partir des caractéristiques des transactions. Ce travail s'inscrit dans un cadre particulièrement difficile : l'enseignant nous a souligné que ce problème est très complexe à apprendre.

Plusieurs objectifs secondaires viennent compléter cette démarche. D'une part, nous cherchons à étudier de manière systématique l'impact du déséquilibre de classes sur les performances des modèles, en montrant par exemple qu'un classifieur peut obtenir une excellente exactitude globale tout en échouant presque totalement à détecter la fraude. D'autre part, nous évaluons différentes stratégies de traitement du déséquilibre, telles que les approches cost-sensitive (pondération des classes dans la fonction de coût), les méthodes de sur-échantillonnage (SMOTE, ADASYN) et certaines approches de détection d'anomalies, afin d'analyser dans quelle mesure ces techniques permettent d'améliorer la détection de la classe minoritaire malgré la difficulté intrinsèque du problème.

5.3 Jeu de données et caractéristiques

Le jeu de données étudié regroupe des transactions financières, chaque ligne correspondant à une opération individuelle et chaque colonne décrivant une caractéristique de cette opération. Les variables explicatives comprennent notamment des informations chiffrées (montants, indicateurs temporels, variables dérivées), tandis que la variable cible indique si la transaction est légitime et acceptée (classe 0) ou frauduleuse (classe 1).

Le jeu de données est de très grande dimension, avec 4 646 773 transactions décrites par 23 variables.

5.4 Déséquilibre des classes

La distribution de la variable cible `FlagImpaye` met en évidence un déséquilibre extrême entre les deux classes. Sur un total de 4 646 773 transactions, 4 616 778 sont des opérations légitimes (classe 0) contre seulement 29 995 transactions frauduleuses (classe 1), soit un taux de fraude d'environ 0,65 %. La figure ci-dessous illustre cette répartition : la part des non-fraudes ($\approx 99,4\%$) occupe quasiment tout le graphique, tandis que la part de fraudes ($\approx 0,6\%$) apparaît comme une fine bande.

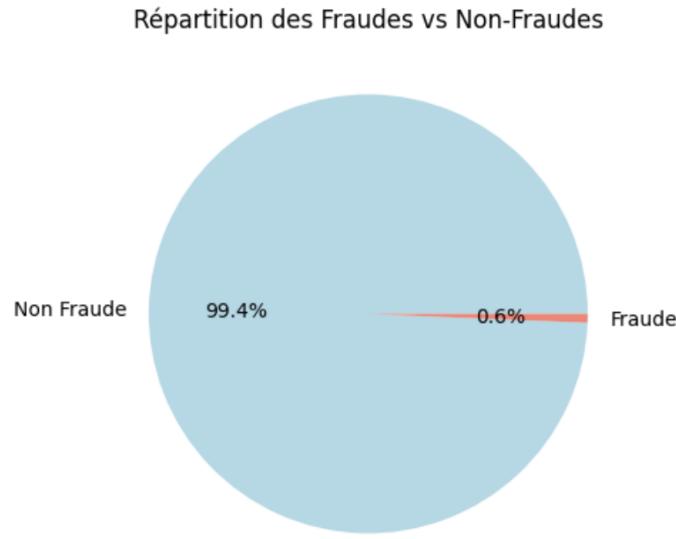


FIGURE 7 – Répartition des classes

6 Méthodologie

6.1 Notations

Notations de base :

- X : matrice des variables explicatives (toutes les colonnes utilisées comme features).
- y : vecteur cible, ici `FlagImpaye`, avec $y_i = 1$ si la transaction i est frauduleuse, $y_i = 0$ sinon.
- Jeu train : utilisé pour apprendre les modèles.
- Jeu test : utilisé pour évaluer les modèles (jamais vu à l'entraînement).

Pour chaque transaction i :

- x_i : vecteur de caractéristiques.
- \hat{p}_i : probabilité prédite que x_i soit une fraude.
- \hat{y}_i : prédiction finale 0 / 1.

Coûts et poids de classes : On distingue 4 cas :

- TP : vraie fraude bien détectée.
- TN : vraie non-fraude bien détectée.
- FP : non-fraude prédite comme fraude (faux positif).
- FN : fraude laissée passer (faux négatif).

On peut associer des coûts (c_{TN} , c_{FP} , c_{FN} , c_{TP}) dans une matrice :

$$\mathbf{C} = \begin{pmatrix} c_{TN} & c_{FP} \\ c_{FN} & c_{TP} \end{pmatrix}.$$

En pratique, on n'écrit pas explicitement tous ces coûts, mais on pèse plus fort la classe fraude.

Fonction de perte : Pour une transaction i , la perte logistique (binary cross-entropy) est :

$$\mathcal{L}_{\log}(y_i, \hat{p}_i) = -(y_i \log(\hat{p}_i) + (1 - y_i) \log(1 - \hat{p}_i)).$$

Sans pondération, la perte moyenne est :

$$\mathcal{L} = \frac{1}{n} \sum_{i=1}^n \mathcal{L}_{\log}(y_i, \hat{p}_i).$$

Avec poids de classes (pour mieux prendre en compte la fraude dans notre cas) :

$$\mathcal{L}_{\text{pond}} = \frac{1}{n} \sum_{i=1}^n w_{y_i} \mathcal{L}_{\log}(y_i, \hat{p}_i),$$

où $w_{y_i} = w_1$ si $y_i = 1$ (fraude), et w_0 sinon.

Seuil de décision : Les modèles (logistique, XGBoost, etc.) donnent une probabilité \hat{p}_i . Pour décider 0 ou 1, on choisit un seuil t :

$$\hat{y}_i = \begin{cases} 1 & \text{si } \hat{p}_i \geq t, \\ 0 & \text{si } \hat{p}_i < t. \end{cases}$$

— Par défaut, $t = 0.5$.

6.2 Mesures de performance

La précision (precision) : La précision mesure la proportion de transactions prédites comme frauduleuses qui sont réellement frauduleuses.

En notant VP les vrais positifs (fraudes correctement détectées) et FP les faux positifs (transactions légitimes à tort signalées comme fraude), on a :

$$\text{Précision} = \frac{VP}{VP + FP}$$

Le rappel (recall ou sensibilité) : Le rappel mesure la capacité du modèle à retrouver les fraudes parmi toutes les fraudes réelles.

En notant FN les faux négatifs (fraudes non détectées), on obtient :

$$\text{Rappel} = \frac{VP}{VP + FN}$$

Le F1-score : Le F1-score est la moyenne harmonique entre précision et rappel, ce qui pénalise fortement les situations où l'une est élevée et l'autre faible.

Il est défini par :

$$F_1 = \frac{2 \times \text{Précision} \times \text{Rappel}}{\text{Précision} + \text{Rappel}}$$

L'AUC-ROC (Area Under the ROC Curve) : L'AUC-ROC mesure le pouvoir de discrimination global du modèle, c'est-à-dire sa capacité à distinguer systématiquement les transactions frauduleuses des transactions légitimes, indépendamment d'un seuil de décision particulier.

Une AUC proche de 1 indique un très bon pouvoir discriminant, tandis qu'une AUC proche de 0,5 correspond à un modèle à peine meilleur que le hasard.

Limites de l'accuracy : L'accuracy est peu informative dans un contexte fortement déséquilibré, comme la fraude où la grande majorité des transactions sont légitimes. Un modèle qui prédirait « non fraude » pour toutes les transactions pourrait ainsi atteindre une accuracy très élevée tout en ne détectant aucune fraude, ce qui est inacceptable opérationnellement.

Pour ces raisons, nous justifions un focus sur **le F1-score** de la classe fraude comme indicateur principal de performance, complété par la précision et le rappel de cette même classe pour analyser le type d'erreurs commises.

6.3 Familles de modèles

Nous avons exploré plusieurs familles de modèles de classification adaptées à la détection de fraude, que nous détaillerons plus loin dans le rapport, couvrant des approches non paramétriques, linéaires, non linéaires, ensemblistes et non supervisées.

Modèles non paramétriques : K plus proches voisins (KNN)

Le K plus proches voisins (KNN) est un classifieur non paramétrique qui ne suppose pas de relation fonctionnelle particulière entre les variables explicatives et la probabilité de fraude.

Pour une nouvelle transaction, il recherche ses K plus proches voisins dans l'espace des variables et lui attribue la classe majoritaire, ce qui en fait un modèle simple à comprendre mais sensible au choix de K et au prétraitement des données.

Cette approche exploite directement la notion de similarité locale entre transactions, permet de modéliser des frontières de décision potentiellement non linéaires sans hypothèse forte sur la distribution des données, et reste très facile à expliquer côté métier (« cette transaction ressemble à telles transactions étiquetées fraude/non fraude »).

Modèles linéaires : régression logistique

La régression logistique modélise la probabilité de fraude comme une combinaison linéaire des variables explicatives, passée dans une fonction logistique, ce qui en fait un modèle simple et bien maîtrisé.

Historiquement, elle est l'un des modèles les plus utilisés en détection de fraude, notamment dans le secteur bancaire et pour les transactions par carte, ce qui en fait une référence naturelle, à la fois interprétable et facilement analysable côté métier.

Modèles non linéaires supervisés : forêts aléatoires

Les forêts aléatoires (Random Forest) agrègent de nombreux arbres de décision construits sur des échantillons bootstrap et des sous-ensembles aléatoires de variables, selon le principe du bagging.

Cette agrégation réduit la variance tout en capturant des relations non linéaires et des interactions, avec une bonne robustesse au bruit et aux outliers.

Nous les retenons comme modèles non linéaires généralistes, performants sur données tabulaires et capables d'exploiter des structures complexes typiques des comportements de fraude.

Méthodes ensemblistes par boosting : XGBoost

Les méthodes de boosting de gradient construisent une suite d'arbres de décision, où chaque nouvel arbre se concentre sur les erreurs commises par les précédents, de façon à réduire progressivement la fonction de perte.

XGBoost met en œuvre ce principe avec de nombreuses optimisations, ce qui en fait un algorithme à la fois rapide et performant pour les données tabulaires.

Dans ce travail, nous le retenons comme candidat principal en termes de performance, car les méthodes de gradient boosting, et XGBoost en particulier, sont souvent parmi les plus compétitives pour la détection de fraude sur données structurées.

Méthodes non supervisées / détection d'anomalies : k-means

Nous testons également une approche non supervisée basée sur le clustering k-means, couramment utilisée pour la détection d'anomalies en contexte de fraude.

L'idée est de regrouper les transactions en clusters de profils typiques, puis de considérer comme suspectes celles qui sont les plus éloignées de ces profils, ce qui permet de repérer des comportements atypiques sans s'appuyer directement sur les labels de fraude.

6.4 Techniques de traitement du déséquilibre

Pour tenir compte du fort déséquilibre entre transactions légitimes et frauduleuses, nous avons recours à plusieurs stratégies de traitement du déséquilibre.

Approches cost-sensitive

Nous utilisons des approches cost-sensitive, qui modifient la fonction de coût pour pénaliser davantage les erreurs sur la classe fraude.

Concrètement, cela passe par la pondération des classes via le paramètre `class_weight` dans certains modèles scikit-learn (par exemple régression logistique et Random Forest), ainsi que par le paramètre `scale_pos_weight` dans XGBoost, qui augmente le poids de la classe minoritaire dans l'optimisation.

Sur-échantillonnage de la classe minoritaire

Nous appliquons également des techniques de sur-échantillonnage pour enrichir la représentation de la classe fraude dans les données d'apprentissage.

Deux méthodes ont été testées : SMOTE, qui génère des exemples synthétiques en interpolant entre voisins de la classe minoritaire, et ADASYN, qui adapte la génération d'exemples synthétiques en se concentrant sur les zones où la minorité est la plus difficile à apprendre.

Sous-échantillonnage de la classe majoritaire

Nous expérimentons aussi un sous-échantillonnage aléatoire de la classe majoritaire, consistant à réduire le nombre de transactions légitimes pour rapprocher les effectifs des deux classes.

Cette approche permet de rééquilibrer rapidement les données, au prix d'une possible perte d'information sur certains profils de clients légitimes.

Ajustement des seuils de décision

Enfin, nous ajustons le seuil de décision utilisé pour convertir les probabilités en prédictions binaires.

Au lieu de conserver le seuil par défaut à 0,5, nous recherchons, sur une grille allant de 0,01 à 0,99, le seuil maximisant le F1-score de la classe fraude, afin d'optimiser directement le compromis précision / rappel sur la classe d'intérêt.

6.5 Approches spécifiques implémentées

Pour synthétiser les différentes combinaisons entre modèles et techniques de traitement du déséquilibre effectivement mises en œuvre dans ce travail, nous résumons ci-dessous les principales approches testées.

Famille de modèle	Variante / pipeline principal	Techniques de déséquilibre associées
Régression logistique	Normalisation → régression logistique	<code>class_weight</code> (balanced), ADASYN
Random Forest	Random Forest avec tuning des hyperparamètres	<code>class_weight</code> (balanced, {0 : 1, 1 : 10}, {0 : 1, 1 : 20}), random undersampling
XGBoost	XGBoost sur variables prétraitées	<code>scale_pos_weight</code> (10, 20, ...), recherche de seuil optimisant le F1 fraude
KNN	KNN sur échantillon de données normalisées	SMOTE
K-means (anomalies)	Clustering k-means, score = distance au centroïde le plus proche	Seuil sur le score d'anomalie pour définir la classe fraude

7 Expériences

7.1 Protocole expérimental

Le jeu de données étudié est de très grande dimension, avec plusieurs millions de transactions et un fort déséquilibre entre opérations légitimes et frauduleuses.

Jeu de données	n	d	Nb fraude	Taux de fraude
Complet	4 646 773	23	29 995	0,65 %

Le découpage entre apprentissage et test suit un split temporel afin de respecter la nature chronologique des données et de se rapprocher d'un scénario de détection en production, conformément aux consignes du sujet.

Les transactions datées du 1^{er} février 2017 au 31 août 2017 constituent l'ensemble d'apprentissage (**3 888 468 observations**), tandis que celles du 1^{er} septembre 2017 au 30 novembre 2017 forment l'ensemble de test (**737 068 observations**).

7.2 Choix des variables

À partir des 23 variables initiales, nous avons construit un jeu de variables explicatives en excluant explicitement certaines colonnes non pertinentes pour la prédiction en ligne. La variable cible `FlagImpaye` n'est évidemment pas utilisée comme entrée du modèle. Nous retirons également `CodeDecision`, qui contient une information de décision a posteriori et ne serait pas disponible au moment de la prédiction, ainsi que `IDAvisAutorisationCheque` et `ZIBZIN`, qui sont des identifiants techniques ne portant pas d'information comportementale exploitable pour la modélisation.

Enfin, la variable `DateTransaction` est exclue des features : elle a déjà été utilisée pour définir le découpage temporel entre apprentissage et test, et sa présence brute dans le modèle pourrait introduire des fuites temporelles ou être difficile à interpréter sous forme numérique.

Les variables explicatives retenues correspondent donc à l'ensemble des colonnes restantes, c'est-à-dire $23 - 5 = 18$ variables, utilisées comme entrées des différents modèles, après un prétraitement minimal consistant à convertir les variables quantitatives textuelles en format numérique (remplacement des virgules par des points, typage numérique), afin de garantir une cohérence de typage et de permettre l'utilisation directe des algorithmes de classification.

7.3 Algorithmes testés

Dans cette section, nous décrivons plus en détail les algorithmes de classification testés ainsi que la manière dont ils ont été adaptés au contexte de fort déséquilibre entre transactions légitimes et frauduleuses.

Pour chaque modèle, nous commençons par une version « brute », entraînée sans traitement spécifique du déséquilibre, qui sert de baseline. Nous introduisons ensuite des variantes intégrant des techniques de rééquilibrage et nous évaluons systématiquement leur impact sur le F1-score de la classe fraude, qui constitue notre métrique principale de performance.

7.3.1 KNN

Le K plus proches voisins (KNN) est un classifieur non paramétrique et « à base d’instances » : il ne construit pas de modèle explicite, mais classe chaque nouvelle transaction en fonction de ses K voisins les plus proches dans l’espace des variables.

Pour une transaction x , on commence par calculer la distance $d(x, x_i)$ entre x et chaque observation d’apprentissage x_i , par exemple avec la distance euclidienne :

$$d(x, x_i) = \sqrt{\sum_{j=1}^d (x_j - x_{i,j})^2}$$

On identifie ensuite l’ensemble $\mathcal{N}_K(x)$ des K plus proches voisins, puis on attribue à x la classe majoritaire parmi ces voisins (vote à la majorité) :

$$\hat{y}(x) = \arg \max_{c \in \{0,1\}} \sum_{x_i \in \mathcal{N}_K(x)} \mathbf{1}_{\{y_i=c\}}$$

KNN brut

Dans notre étude, nous commençons par un KNN brut (sans traitement spécifique du déséquilibre) qui sert de baseline. Comme le KNN repose sur le calcul de distances entre chaque nouvelle observation et l’ensemble des points d’apprentissage, son coût de calcul devient rapidement important sur un jeu de données de plusieurs millions de transactions.

Pour le rendre exploitable, nous l’entraînons donc sur un sous-échantillon aléatoire de l’ensemble d’apprentissage, limité à 400 000 transactions, et nous l’évaluons sur un sous-échantillon de 100 000 transactions issues de l’ensemble de test, en conservant la même définition de la cible et des variables explicatives. Les variables numériques sont préalablement standardisées au sein d’un pipeline (standardisation puis KNN avec $\mathbf{K} = 5$ et pondération par la distance), afin de garantir que toutes les dimensions contribuent de manière comparable au calcul des distances.

Les résultats obtenus avec le KNN brut sur le sous-échantillon de test fournissent une première indication des limites de ce modèle dans un contexte fortement déséquilibré.

Classe	Précision	Rappel	F1-score	Accuracy globale
0 (non-fraude)	0,991	1,000	0,995	
1 (fraude)	0,158	0,010	0,020	0,991

On constate que l’accuracy globale reste très élevée (0,991), mais que la performance sur la classe fraude est extrêmement faible ($F1 \approx 0,02$, rappel $\approx 1\%$), ce qui illustre bien le caractère trompeur de l’accuracy en présence d’un fort déséquilibre de classes et motive l’utilisation de techniques de rééquilibrage pour améliorer le F1-score fraude.

KNN avec SMOTE

Pour améliorer le KNN dans ce contexte très déséquilibré, nous testons une seconde variante qui combine sur-échantillonnage SMOTE et KNN au sein d’un même pipeline.

SMOTE (Synthetic Minority Over-sampling Technique) est une méthode de suréchantillonnage de la classe minoritaire qui ne se contente pas de dupliquer les fraudes existantes, mais génère de nouveaux exemples synthétiques en interpolant entre des fraudes voisines.

L'idée est de densifier la région de la classe fraude dans l'espace des variables, afin que les algorithmes de classification (comme KNN) voient davantage d'exemples de ce type et apprennent une frontière de décision moins biaisée vers la classe majoritaire.

Schématiquement, pour chaque observation frauduleuse x , SMOTE :

- trouve ses k plus proches voisins dans la classe minoritaire ;
- choisit au hasard un voisin x_{voisin} parmi ces k voisins ;
- génère un nouveau point synthétique le long du segment (x, x_{voisin}) selon

$$x_{\text{synt}} = x + \lambda \cdot (x_{\text{voisin}} - x), \quad \lambda \in (0, 1)$$

où λ est un nombre aléatoire uniforme. Répété plusieurs fois, ce procédé crée un nuage de nouvelles fraudes « réalistes » autour des fraudes existantes, sans se limiter à de simples copies.

Dans notre travail, SMOTE est appliqué uniquement sur l'ensemble d'apprentissage, puis les variables sont standardisées et un KNN ($K = 5$, pondération par la distance) est entraîné sur ce jeu rééquilibré, en conservant les mêmes sous-échantillons train/test que pour le KNN brut afin de mesurer proprement l'impact du sur-échantillonnage sur le F1-score de la classe fraude.

Les performances de la variante KNN + SMOTE sont résumées ci-dessous.

Classe	Précision	Rappel	F1-score	Accuracy globale
0 (non-fraude)	0,993	0,830	0,904	
1 (fraude)	0,017	0,335	0,032	0,826

On observe que l'accuracy globale diminue fortement (de 0,991 à 0,826), car le modèle commence à classer davantage de transactions comme frauduleuses, ce qui dégrade la performance sur la classe majoritaire. En revanche, le rappel de la classe fraude passe d'environ 1 % à 33,5 % et le F1 fraude progresse de 0,020 à 0,032 : le modèle détecte nettement plus de fraudes, mais au prix d'une précision extrêmement faible (1,7 %), ce qui signifie beaucoup de faux positifs, illustrant bien le compromis induit par SMOTE dans ce cas.

7.3.2 Régression Logistique

La régression logistique binaire modélise directement la probabilité qu'une transaction soit frauduleuse ($Y = 1$) à partir d'une combinaison linéaire des variables explicatives x . On écrit :

$$P(Y = 1 | x) = \sigma(w^\top x + b)$$

où σ est la fonction logistique (sigmoïde) définie par :

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Le modèle fournit donc, pour chaque transaction, une probabilité de fraude comprise entre 0 et 1.

En pratique, on applique ensuite un seuil de décision (par défaut 0,5) : si $P(Y = 1 | x) > 0,5$, la transaction est classée comme fraude, sinon comme non-fraude.

Historiquement, la régression logistique est très utilisée en détection de fraude (notamment bancaire et cartes de paiement) car elle combine simplicité, interprétabilité et bonne performance de base sur des données tabulaires.

Régression logistique brute

Nous commençons par entraîner une régression logistique standard sur l'ensemble complet d'apprentissage, sans traitement spécifique du déséquilibre. Le modèle est intégré dans un pipeline comprenant une standardisation des variables, puis une régression logistique avec régularisation L2 (Ridge) et le solveur `lbfgs`, adapté aux grands jeux de données et aux problèmes de classification binaire.

La régularisation L2 ajoute une pénalité de complexité à la fonction de coût de la forme

$$\mathcal{L}_{\text{reg}}(w) = \mathcal{L}_{\text{logistique}}(w) + \lambda \sum_j w_j^2$$

où $\lambda > 0$ contrôle l'intensité de la pénalisation et $\sum_j w_j^2$ correspond à la norme L2 des coefficients du modèle, ce qui décourage les poids de grande amplitude, limite le sur-apprentissage et améliore la généralisation.

Les résultats obtenus sur l'ensemble de test sont les suivants :

Classe	Précision	Rappel	F1-score	Accuracy globale
0 (non-fraude)	0,991	1,000	0,996	
1 (fraude)	0,241	0,002	0,004	0,991

On observe une accuracy globale très élevée (0,991), et une excellente performance sur la classe non-fraude ($F1 \approx 0,996$), mais la détection de la fraude reste quasi nulle : le rappel de la classe 1 est d'environ 0,2 % et le F1-score fraude tombe à 0,004. Ce comportement illustre le biais fort d'un modèle non adapté au déséquilibre et motive l'introduction de versions cost-sensitive (via `class_weight`) et/ou combinées à du sur-échantillonnage pour améliorer le F1-score de la classe fraude.

Régression logistique cost-sensitive

Nous entraînons ensuite une régression logistique cost-sensitive en activant l'option `class_weight="balanced"` dans `LogisticRegression`, toujours avec pénalisation L2 et le solveur `lbfgs` dans le même pipeline de standardisation.

Concrètement, `class_weight="balanced"` calcule automatiquement un poids w_j pour chaque classe j selon la formule

$$w_j = \frac{n_{\text{samples}}}{n_{\text{classes}} \cdot n_j}$$

où n_{samples} est le nombre total d'exemples, n_{classes} le nombre de classes et n_j le nombre d'exemples de la classe j . Ainsi, la classe minoritaire (fraude) reçoit un poids beaucoup plus élevé que la classe majoritaire, ce qui rend les erreurs sur la fraude plus pénalisantes dans la fonction de coût et pousse le modèle à mieux la détecter, sans modifier le jeu de données lui-même.

Les résultats obtenus sur l'ensemble de test sont les suivants :

Classe	Précision	Rappel	F1-score	Accuracy globale
0 (non-fraude)	0,995	0,748	0,854	
1 (fraude)	0,020	0,579	0,039	0,746

Sur le jeu de test, ce modèle obtient une accuracy d'environ 0,746, avec de très bons résultats pour la classe non-fraude (précision 0,995, rappel 0,748, F1-score 0,854) et, surtout, une forte amélioration du rappel fraude qui passe à 0,579, au prix d'une précision très faible (0,020) et d'un F1-score de 0,039 pour la classe 1. Ce comportement est typique des modèles cost-sensitive en détection de fraude : ils acceptent beaucoup plus de faux positifs pour capter un maximum de transactions frauduleuses, ce qui est souvent souhaitable quand le coût d'une fraude non détectée est beaucoup plus élevé que celui d'un contrôle supplémentaire sur une transaction légitime.

Régression logistique avec ADASYN

Nous testons enfin une régression logistique combinée à un sur-échantillonnage ADASYN (Adaptive Synthetic Sampling), en insérant l'étape ADASYN dans un pipeline imblearn entre la standardisation des variables et la régression logistique L2 avec solveur `lbfgs`. ADASYN est une variante adaptative de SMOTE : au lieu de générer des exemples synthétiques de la classe minoritaire de manière uniforme, il identifie (via un k-plus proches voisins) les régions de l'espace de caractéristiques où les exemples minoritaires sont entourés de nombreux exemples majoritaires, puis génère davantage d'exemples synthétiques dans ces zones difficiles à apprendre. L'idée est de déplacer la frontière de décision vers les régions où la minorité est la plus complexe ou la plus ambiguë, afin de réduire le biais dû au déséquilibre tout en enrichissant localement la structure de la classe fraude.

Les résultats obtenus sur l'ensemble de test sont les suivants :

Classe	Précision	Rappel	F1-score	Accuracy globale
0 (non-fraude)	0,995	0,729	0,842	
1 (fraude)	0,019	0,594	0,037	0,728

On observe une accuracy légèrement inférieure à la version cost-sensitive seule (0,728 contre 0,746), mais avec des métriques très proches : la classe non-fraude reste bien modélisée (F1-score 0,842) et le rappel de la classe fraude atteint 0,594, toujours au prix d'une précision très faible (0,019) et d'un F1-score de 0,037. Ce résultat illustre un point important des méthodes de sur-échantillonnages avancés comme ADASYN : elles peuvent améliorer localement l'apprentissage de la minorité, mais dans un cadre déjà très pénalisé vers la fraude (ici via l'architecture logistique + L2), le compromis précision/rappel reste dominé par la difficulté intrinsèque du problème et par le choix d'un seuil de décision par défaut à 0,5.

7.3.3 Random Forest

Un arbre de décision est un modèle qui prend des décisions en suivant une succession de questions du type « SI ... ALORS ... » organisées en forme d'arbre. À chaque nœud,

il choisit une variable X_j et un seuil s qui séparent le mieux les classes en maximisant, par exemple, la réduction d'impureté de Gini

$$\Delta G = G(\text{noeud}) - \left(\frac{n_{\text{gauche}}}{n} G(\text{gauche}) + \frac{n_{\text{droite}}}{n} G(\text{droite}) \right)$$

où $G(t) = 1 - \sum_k p_{k,t}^2$ est l'indice de Gini au nœud t , $p_{k,t}$ la proportion de la classe k dans ce nœud, et n_{gauche} , n_{droite} les effectifs après la coupure. En répétant ce processus jusqu'à un critère d'arrêt (profondeur max, nombre min d'exemples, etc.), on obtient des feuilles qui prédisent la classe majoritaire du nœud.

Un random forest construit ensuite un ensemble de B arbres $\{T_b\}_{b=1}^B$ sur des échantillons bootstrapés du jeu d'apprentissage et avec un sous-ensemble aléatoire de variables testé à chaque nœud. Pour une nouvelle observation x , chaque arbre fournit une prédiction $\hat{y}_b(x)$, et la prédiction finale est obtenue par vote majoritaire

$$\hat{y}_{\text{RF}}(x) = \arg \max_k \sum_{b=1}^B \mathbf{1}(\hat{y}_b(x) = k),$$

ou, de manière équivalente, par la moyenne des probabilités de classe $\hat{p}_{\text{RF}}(Y = k \mid x) = \frac{1}{B} \sum_{b=1}^B \hat{p}_b(Y = k \mid x)$ si chaque arbre produit des scores probabilistes. Ce schéma de bagging (bootstrap aggregating) permet de réduire la variance par rapport à un arbre unique, en exploitant le fait que des estimateurs instables comme les arbres deviennent beaucoup plus stables et performants lorsqu'on les moyenne sur de nombreux tirages de données et de variables.

Random Forest brut

Pour commencer, nous définissons une forêt aléatoire de base qui nous sert de modèle de référence sans traitement spécifique du déséquilibre, mais avec des garde-fous simples contre le sur-apprentissage. Le modèle utilise `n_estimators = 100` arbres, un choix standard qui offre en pratique un bon compromis entre stabilité du vote majoritaire et temps de calcul sur l'ensemble complet de transactions. Nous fixons la profondeur des arbres à `max_depth = 15` afin de limiter la complexité de chaque arbre, des arbres très profonds peuvent mémoriser le bruit, alors qu'une profondeur modérée augmente légèrement le biais mais réduit la variance globale, ce qui est souhaitable sur un problème de fraude bruité et déséquilibré. Nous imposons également `min_samples_leaf = 5`, ce qui empêche de créer des feuilles contenant moins de cinq observations ; cela évite des règles reposant sur quelques cas isolés seulement et constitue un second levier simple pour contrôler le sur-apprentissage. Enfin, `n_jobs = -1` autorise l'utilisation de tous les coeurs disponibles.

Les performances obtenues sur le jeu de test sont les suivantes :

Classe	Précision	Rappel	F1-score	Accuracy globale
0 (non-fraude)	0,991	1,000	0,996	
1 (fraude)	0,695	0,018	0,035	0,991

Nous retrouvons une accuracy globale très élevée (0,991) et un comportement quasi parfait sur la classe majoritaire ($F1 \approx 0,996$), ce qui montre que la forêt apprend très bien les transactions non frauduleuses. En revanche, la détection de fraude reste très faible : le

rappel de la classe 1 n'est que de 1,8 %, malgré une précision relativement élevée (0,695), ce qui se traduit par un F1-score fraude d'environ 0,035. Ce RF brut présente donc le même symptôme que la régression logistique de base : il est fortement biaisé vers la classe non-fraude et nous sert principalement de baseline pour évaluer l'apport de versions adaptées au déséquilibre (poids de classes, rééchantillonnage).

Random Forest balanced

Nous testons ensuite une forêt aléatoire cost-sensitive en activant l'option `class_weight="balanced"` dans `RandomForestClassifier`, tout en conservant la même structure de modèle que pour le RF brut (`n_estimators=100, max_depth=15, min_samples_leaf=5, n_jobs=-1`). Le paramètre `class_weight="balanced"` ajuste automatiquement les poids comme expliqué précédemment. La classe fraude, beaucoup plus rare, reçoit donc un poids nettement plus élevé : les erreurs de classification sur cette classe deviennent plus pénalisantes dans la fonction de coût interne, ce qui pousse la forêt à produire davantage de feuilles orientées vers la détection de fraude, au prix d'une dégradation attendue de l'accuracy globale.

Les performances obtenues sur le jeu de test sont les suivantes :

Classe	Précision	Rappel	F1-score	Accuracy globale
0 (non-fraude)	0,994	0,825	0,902	
1 (fraude)	0,024	0,481	0,046	0,822

Nous observons une baisse de l'accuracy (0,822 contre 0,991 pour le RF brut) et une nette dégradation des performances sur la classe non-fraude (F1-score 0,902), ce qui traduit une augmentation importante des faux positifs. En contrepartie, le rappel de la classe fraude progresse de manière significative (de 0,018 à 0,481), au prix d'une précision très faible (0,024) et d'un F1-score fraude qui reste modeste ($\approx 0,046$). Ce comportement illustre bien l'effet de la pondération des classes : le modèle devient beaucoup plus « agressif » dans la détection de la fraude, en acceptant de signaler un grand nombre de transactions légitimes comme suspectes, ce qui peut être acceptable si le coût d'une fraude manquée est bien plus élevé qu'un contrôle supplémentaire sur une transaction normale.

Random Forest avec sous-échantillonnage

Nous testons ensuite une forêt aléatoire avec undersampling aléatoire de la classe majoritaire, en insérant un `RandomUnderSampler` dans un pipeline imblearn avant le classifieur Random Forest (`n_estimators=100, max_depth=15, min_samples_leaf=5`). Le `RandomUnderSampler` réduit le nombre d'exemples de la classe non-fraude en en supprimant une partie de manière aléatoire, jusqu'à obtenir un jeu d'entraînement beaucoup plus équilibré ; la forêt est donc entraînée sur un dataset où la fraude est relativement fréquente, ce qui la pousse à apprendre une frontière de décision plus sensible à cette classe, au prix d'une perte d'information sur une partie des transactions légitimes.

Les performances obtenues sur le jeu de test sont les suivantes :

Classe	Précision	Rappel	F1-score	Accuracy globale
0 (non-fraude)	0,996	0,652	0,788	
1 (fraude)	0,018	0,699	0,034	0,652

Nous constatons une chute très forte de l'accuracy (0,652) et une nette détérioration du comportement sur la classe non-fraude (rappel $\approx 0,65$, F1-score 0,788), signe que le modèle produit beaucoup de faux positifs. En revanche, le rappel fraude devient très élevé ($\approx 0,70$), mais avec une précision extrêmement faible ($\approx 0,018$), ce qui maintient le F1-score fraude à un niveau modeste ($\approx 0,034$). Cette version Random Forest + undersampling illustre bien le compromis induit par le sous-échantillonnage aléatoire : en sacrifiant une partie de l'information de la classe majoritaire, on obtient un modèle très agressif pour détecter la fraude, mais au prix d'un nombre important d'alertes injustifiées sur des transactions légitimes.

Random Forest cost-sensitive optimisée (poids {0 :1, 1 :20})

Nous construisons ensuite une Random Forest cost-sensitive tunée manuellement, dans laquelle nous renforçons fortement la pénalisation des erreurs sur la classe positive (fraude). Nous augmentons d'abord le nombre d'arbres à `n_estimators = 200` pour stabiliser davantage le vote de la forêt, et nous retirons la contrainte sur la profondeur (`max_depth = None`), tout en durcissant la régularisation via `min_samples_leaf = 10`, ce qui impose des feuilles plus grosses et limite la tendance des arbres profonds à sur-apprendre des motifs très locaux. Surtout, nous remplaçons `class_weight="balanced"` par une pondération manuelle `class_weight={0: 1, 1: 20}` : chaque erreur sur une transaction frauduleuse (classe 1) est 20 fois plus coûteuse qu'une erreur sur une transaction normale (classe 0) dans la fonction de coût interne, ce qui pousse la forêt à être beaucoup plus prudente vis-à-vis des fraudes potentielles.

Les performances obtenues sur le jeu de test sont les suivantes :

Classe	Précision	Rappel	F1-score	Accuracy globale
0 (non-fraude)	0,992	0,994	0,993	
1 (fraude)	0,119	0,098	0,107	0,986

Par rapport au RF cost-sensitive simple, nous récupérons une accuracy globale élevée (0,986) et un comportement presque parfait sur la classe non-fraude (F1 $\approx 0,993$), tout en améliorant sensiblement le F1-score fraude ($\approx 0,107$ contre $\approx 0,046$ précédemment). Le rappel fraude ($\approx 0,10$) reste modeste, mais il s'accompagne d'une précision nettement meilleure ($\approx 0,12$) que dans les versions très agressives (`class_weight="balanced"` ou undersampling), ce qui montre que cette Random Forest cost-sensitive tunée par pondérations explicites {0 :1, 1 :20} offre un compromis plus équilibré entre stabilité globale du modèle et détection utile des fraudes.

Random Forest cost-sensitive optimisée (poids {0 :1, 1 :10})

Nous retenons également une Random Forest cost-sensitive tunée avec une deuxième configuration après plusieurs essais. Nous augmentons d'abord le nombre d'arbres à `n_estimators` = 300 afin de stabiliser davantage le vote de la forêt, tout en conservant une profondeur maximale `max_depth` = 15 pour garder des arbres riches mais contrôlés sur un problème complexe comme la fraude. Nous assouplissons légèrement la régularisation structurelle en fixant `min_samples_leaf` = 2, ce qui autorise des feuilles un peu plus petites et donc des règles plus fines que dans la configuration `min_samples_leaf` = 10, et nous explicitons le choix `max_features` = "sqrt", qui est le réglage classique pour les forêts de classification : à chaque noeud, seule la racine carrée du nombre total de variables est considérée pour le split, ce qui introduit de la diversité entre arbres et réduit la variance de l'ensemble. Côté coût, nous conservons une pénalisation asymétrique mais un peu moins extrême que précédemment, avec `class_weight` = {0 : 1, 1 : 10} : une erreur sur une transaction frauduleuse reste dix fois plus coûteuse qu'une erreur sur une transaction légitime, ce qui maintient le modèle orienté vers la détection de fraude tout en limitant l'explosion des faux positifs.

Les performances obtenues sur le jeu de test sont les suivantes :

Classe	Précision	Rappel	F1-score	Accuracy globale
0 (non-fraude)	0,992	0,998	0,995	
1 (fraude)	0,190	0,065	0,097	0,989

Nous obtenons une accuracy globale très élevée (0,989) et un F1-score quasi parfait pour la classe non-fraude (0,995), ce qui montre que cette configuration reste très fiable sur la majorité. Pour la classe fraude, le F1-score ($\approx 0,097$) est légèrement inférieur à celui de la configuration fortement pénalisée {0 :1, 1 :20}, mais avec une précision un peu meilleure (0,19) et un rappel plus faible (0,065), ce qui correspond à un modèle plus conservateur dans la génération d'alertes.

7.3.4 K-means

Nous testons K-means comme méthode d'anomalie non supervisée, car il est fréquemment utilisé en détection de fraude pour repérer des comportements atypiques sans utiliser directement les labels lors de l'apprentissage.

K-means cherche à partitionner les données en K groupes en minimisant la somme des distances au centroïde de chaque cluster. Mathématiquement, il minimise la fonction objective

$$J = \sum_{k=1}^K \sum_{x_i \in C_k} \|x_i - \mu_k\|^2$$

où C_k est l'ensemble des points du cluster k et μ_k son centroïde. L'algorithme alterne deux étapes :

- **Affectation** : chaque point x_i est assigné au cluster dont le centroïde est le plus proche,

$$\text{cluster}(x_i) = \arg \min_k \|x_i - \mu_k\|,$$

- **Mise à jour** : chaque centroïde μ_k devient la moyenne des points de son cluster.

Ces étapes sont répétées jusqu'à stabilisation des affectations ou de J .

Dans notre cas, nous standardisons d'abord les variables, puis ajustons un K-means avec $K = 2$ clusters, interprétés comme « comportement normal » vs « comportement atypique ». Pour chaque transaction, nous calculons la distance au centroïde le plus proche

$$d(x) = \min_k \|x - \mu_k\|,$$

et nous utilisons cette distance comme score d'anomalie : plus $d(x)$ est grande, plus la transaction est suspecte. Nous fixons ensuite un seuil basé sur le quantile 99,5 % des distances d'entraînement (0,995) : les 0,5 % de points les plus éloignés sont étiquetés « fraude », ce qui revient à détecter des outliers par rapport au cœur du comportement normal.

Les performances obtenues sur le jeu de test sont les suivantes :

Classe	Précision	Rappel	F1-score	Accuracy globale
0 (non-fraude)	0,992	0,994	0,993	
1 (fraude)	0,075	0,057	0,064	0,986

Nous obtenons une accuracy élevée (0,986) et un très bon comportement sur la classe non-fraude, mais le F1-score fraude reste faible ($\approx 0,064$), avec un rappel de l'ordre de 5–6 % seulement. Ce résultat est cohérent avec la logique de K-means : le modèle capture bien la structure dominante du comportement normal, mais une partie importante des fraudes n'apparaît pas comme des outliers très éloignés des centroïdes, ce qui limite son intérêt comme solution principale (sauf dans le cas d'une optimisation).

7.3.5 XGBoost

XGBoost est une implémentation optimisée de gradient boosting sur arbres de décision, où le modèle est construit de façon additive : on ajoute des arbres successifs pour corriger les erreurs des précédents. Le modèle final s'écrit comme une somme de T arbres f_t (de type CART)

$$\hat{y}_i = \sum_{t=1}^T f_t(x_i),$$

puis ces scores sont transformés en probabilités (par exemple via une sigmoïde) pour la classification binaire.

À chaque itération t , XGBoost ajoute un nouvel arbre f_t en minimisant une fonction objectif régularisée

$$\mathcal{L}^{(t)} = \sum_i \ell(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t),$$

où ℓ est la perte (log-loss en binaire) et $\Omega(f_t)$ pénalise la complexité de l'arbre (nombre de feuilles, poids des feuilles), ce qui limite le sur-apprentissage. Concrètement, chaque nouvel arbre est ajusté sur les erreurs résiduelles du modèle courant, avec un learning rate qui contrôle l'ampleur de la correction à chaque étape ; on obtient ainsi un ensemble d'arbres peu profonds mais nombreux, particulièrement efficace sur des données tabulaires pour des tâches comme la détection de fraude.

XGBoost brut

Nous commençons par une version brute d’XGBoost qui sert de baseline supervisée, sans traitement explicite du déséquilibre autre que ce que gère l’algorithme par défaut. Nous utilisons un classifieur `XGBClassifier` avec une objective de type `binary:logistic`, adaptée à la classification binaire : le modèle produit un score \hat{y}_i agrégé par les arbres, puis applique une sigmoïde pour obtenir une probabilité de fraude $P(Y = 1 | x_i)$, qui est ensuite seuillée à 0,5 pour prédire la classe. Les hyperparamètres `n_estimators = 300`, `max_depth = 6` et `learning_rate = 0.1` correspondent à des valeurs raisonnables par défaut pour un boosting de gradient sur données tabulaires. Enfin, `subsample = 1.0` et `colsample_bytree = 1.0` indiquent que chaque arbre est entraîné sur toutes les observations et toutes les variables (sans sous-échantillonnage), ce qui simplifie l’interprétation de cette première version brute.

Les performances obtenues sur le jeu de test sont les suivantes :

Classe	Précision	Rappel	F1-score	Accuracy globale
0 (non-fraude)	0,992	1,000	0,996	
1 (fraude)	0,540	0,041	0,076	0,991

Nous obtenons une accuracy globale très élevée (0,991) et un comportement presque parfait sur la classe non-fraude ($F1 \approx 0,996$), ce qui montre qu’XGBoost modélise très bien la majorité. Pour la classe fraude, la précision ($\approx 0,54$) est sensiblement plus élevée que pour les modèles linéaires ou certaines forêts, mais le rappel reste très faible ($\approx 4\%$), ce qui maintient le F1-score fraude à un niveau modeste ($\approx 0,076$). Cette version XGBoost brut se comporte donc comme les autres modèles non adaptés au déséquilibre : elle privilégie la classe majoritaire et sert de point de départ pour introduire ensuite des réglages spécifiques (pondération via `scale_pos_weight`, tuning orienté F1 fraude, ajustement de seuil).

XGBoost cost-sensitive

Nous testons ensuite une version cost-sensitive d’XGBoost, dans laquelle nous répondrons explicitement la classe fraude via le paramètre `scale_pos_weight`. Le modèle conserve la même structure de base (`n_estimators = 300`, `max_depth = 6`, `learning_rate = 0.1`), mais introduit deux sources de régularisation stochastique supplémentaires : `subsample = 0.8` (chaque arbre est entraîné sur 80 % des lignes tirées aléatoirement) et `colsample_bytree = 0.8` (chaque arbre ne voit que 80 % des variables), ce qui réduit le risque de sur-apprentissage et augmente la diversité entre arbres. Le paramètre clé est `scale_pos_weight = 10`, qui multiplie le poids des exemples de la classe positive dans le calcul des gradients ; en pratique, cela revient à dire qu’une erreur sur une transaction frauduleuse compte environ dix fois plus qu’une erreur sur une transaction non frauduleuse, ce qui pousse XGBoost à accorder beaucoup plus d’importance à la minorité sans modifier le jeu de données.

Les performances obtenues sur le jeu de test sont les suivantes :

Classe	Précision	Rappel	F1-score	Accuracy globale
0 (non-fraude)	0,993	0,986	0,989	
1 (fraude)	0,101	0,182	0,130	0,979

Par rapport à la version XGBoost brut, nous observons une légère baisse de l'accuracy (0,979 contre 0,991) et une dégradation des performances sur la classe non-fraude (F1-score 0,989), ce qui traduit l'augmentation des faux positifs. En contrepartie, le F1-score fraude progresse nettement ($\approx 0,13$ contre $\approx 0,076$), grâce à un rappel plus élevé ($\approx 18\%$) au prix d'une précision modeste ($\approx 10\%$). Cette version XGBoost cost-sensitive (`scale_pos_weight = 10`) illustre donc bien l'usage de ce paramètre : il permet de déplacer le modèle vers une détection de fraude plus agressive, en acceptant une dégradation contrôlée des performances sur la classe majoritaire.

XGBoost cost-sensitive avec optimisation du seuil

Nous partons de la configuration cost-sensitive précédente (`scale_pos_weight = 10`) et ajoutons une optimisation du seuil de décision pour maximiser spécifiquement le F1-score de la classe fraude. Au lieu d'utiliser le seuil par défaut 0,5 sur la probabilité prédictive $P(Y = 1 | x)$, nous balayons un ensemble de seuils $t \in [0,01, 0,99]$ et, pour chaque t , nous calculons le F1-score obtenu en prédisant fraude si $P(Y = 1 | x) \geq t$. Nous retenons le seuil t^* qui maximise ce F1-score et l'utilisons ensuite pour générer les prédictions finales.

Les performances obtenues sur le jeu de test avec le seuil optimal $t^* \approx 0,63$ sont les suivantes :

Classe	Précision	Rappel	F1-score	Accuracy globale
0 (non-fraude)	0,992	0,995	0,994	
1 (fraude)	0,172	0,121	0,142	0,987

Par rapport à la version XGBoost cost-sensitive à seuil fixe (0,5), l'accuracy reste très élevée (0,987) et la performance sur la classe non-fraude ne se dégrade quasiment pas, tandis que le F1-score fraude est légèrement amélioré ($\approx 0,142$ contre $\approx 0,130$), grâce à un compromis un peu différent entre précision (0,172) et rappel (0,121). Cette version « XGBoost cost-sensitive avec optimisation du seuil » illustre l'intérêt du threshold moving dans les problèmes déséquilibrés : une fois le modèle entraîné, ajuster le seuil de décision permet de gagner quelques points de F1 sur la minorité sans ré-entraîner le modèle.

XGBoost cost-sensitive fortement pénalisé avec seuil optimisé

Nous testons enfin une configuration XGBoost plus agressive pour la classe fraude, en combinant une pénalisation renforcée (`scale_pos_weight = 20`) avec un recalibrage du seuil pour maximiser le F1-score fraude. Le modèle utilise davantage d'arbres (`n_estimators = 500`) mais moins profonds (`max_depth = 4`) et un learning rate plus faible (`learning_rate = 0,05`), ce qui correspond à une stratégie classique de boosting « lent mais régulier » : de nombreux arbres peu profonds qui corrigent progressivement les erreurs, tout en limitant le sur-apprentissage. Comme pour la version précédente, nous fixons

`subsample = 0.8` et `colsample_bytree = 0.8` pour introduire du hasard ligne/colonnes et améliorer la robustesse, puis nous balayons un ensemble de seuils sur la probabilité prédictive $P(Y = 1 | x)$ afin de choisir le seuil t^* maximisant le F1-score de la classe fraude.

Avec cette configuration, le meilleur seuil trouvé est $t^* \approx 0.72$. Les performances obtenues sur le jeu de test sont :

Classe	Précision	Rappel	F1-score	Accuracy globale
0 (non-fraude)	0,992	0,993	0,993	
1 (fraude)	0,150	0,145	0,147	0,985

Cette version constitue notre meilleur F1-score sur la classe fraude tout en conservant une accuracy globale élevée (0,985) et un excellent comportement sur la classe non-fraude. Elle illustre l'intérêt de combiner une pénalisation forte de la classe positive (`scale_pos_weight = 20`) avec un ajustement fin du seuil de décision lorsque l'objectif principal est d'optimiser le F1-score de la minorité dans un contexte de détection de fraude très déséquilibré.

7.4 Optimisation des hyperparamètres

Dans cette section, nous présentons la façon dont nous avons cherché à améliorer les modèles tout au long du projet, en jouant à la fois sur leurs hyperparamètres, sur la gestion du déséquilibre et sur le choix du seuil de décision. Plutôt que de lancer des recherches massives très coûteuses, nous avons combiné des réglages manuels raisonnés, quelques essais de recherche aléatoire (notamment pour Random Forest et XGBoost) et une optimisation systématique du F1-score de la classe fraude, qui reste la métrique centrale de notre étude.

- **Point de départ : valeurs raisonnables.** Pour chaque modèle (régression logistique, Random Forest, XGBoost, KNN, K-means), nous sommes partis de paramètres proches des valeurs par défaut (profondeur et nombre d'arbres, nombre de voisins, pénalisation L2, nombre de clusters, etc.), ajustés à la marge selon les bonnes pratiques pour données tabulaires déséquilibrées.
- **Compromis biais / variance.**
 - Forêts et XGBoost : ajustement de `max_depth`, `min_samples_leaf`, `n_estimators`, `subsample`, `colsample_bytree` pour éviter des arbres trop profonds tout en gardant un nombre d'arbres suffisant pour stabiliser les prédictions.
 - Régression logistique : choix d'une régularisation L2 standard et d'un solveur adapté.
 - K-means : fixation de $K = 2$ (normal vs atypique), cohérent avec l'objectif détection de fraude.
- **Hyperparamètres liés au déséquilibre.**
 - `class_weight` pour la régression logistique et la Random Forest ("balanced", puis poids explicites comme $\{0 : 1, 1 : 20\}$).
 - `scale_pos_weight` pour XGBoost (valeurs testées 8, 10, 15, 20).

Ces valeurs ont été augmentées ou réduites progressivement en fonction de leur effet sur le F1-score de la classe fraude.

- **Ajustements manuels itératifs.** Autour de chaque configuration de départ, nous avons fait varier un petit nombre d'hyperparamètres (par exemple `n_estimators` 100 → 300 → 500, `max_depth` 4 → 6, `learning_rate` 0,1 → 0,05) et conservé les combinaisons améliorant le F1 de la classe 1 sur les jeux de validation, plutôt que de chercher un optimum exhaustif.
- **RandomizedSearchCV comme complément.**
 - Mise en place d'une `RandomizedSearchCV` pour XGBoost sur un espace restreint (`n_estimators`, `max_depth`, `learning_rate`, `subsample`, `colsample_bytree`, `scale_pos_weight`), avec un scorer basé sur `f1_score(pos_label=1)`, un `n_iter` limité et `cv = 2`.
 - Tentatives analogues envisagées pour la Random Forest.
 - En pratique, ces recherches ont été très coûteuses en temps et en ressources (taille du dataset, GPU/CPU) ; elles n'ont pas abouti dans des délais raisonnables.

8 Résultats

8.1 Présentation des résultats

La figure suivante présente une vue d'ensemble des performances des différentes configurations testées, sous forme de heatmap. Chaque ligne correspond à un modèle ou à une variante précise, et chaque colonne reporte respectivement le F1-score fraude, la précision fraude et le rappel fraude obtenus sur le jeu de test. Les couleurs sont directement proportionnelles au F1-score fraude : plus la case est foncée, meilleur est le compromis précision/rappel pour la détection de la classe positive. Cette représentation permet de comparer visuellement, en un coup d'œil, quelles familles de modèles et quelles stratégies apportent les gains les plus significatifs sur la détection de fraude.

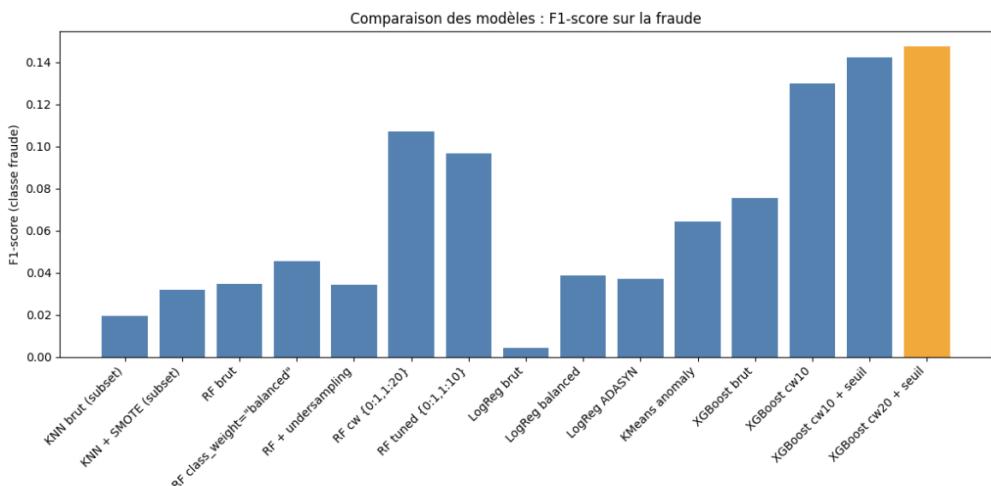


FIGURE 8 – Heatmap des performances des modèles sur la classe fraude

Pour compléter cette vue globale, nous représentons sur la figure suivante le F1-score fraude de chaque modèle sous forme d'histogramme. Chaque barre correspond à une configuration particulière (KNN, régression logistique, Random Forest, XGBoost, avec

ou sans techniques de rééchantillonnage, poids de classes ou optimisation de seuil), et la hauteur de la barre indique directement le F1-score obtenu sur la classe fraude. La barre mise en évidence (en orange) correspond à la meilleure configuration du projet, à savoir le modèle XGBoost cost-sensitive avec forte pénalisation de la classe positive et seuil de décision optimisé, qui atteint un F1-score fraude d'environ 0,15 et se détache clairement des autres approches.

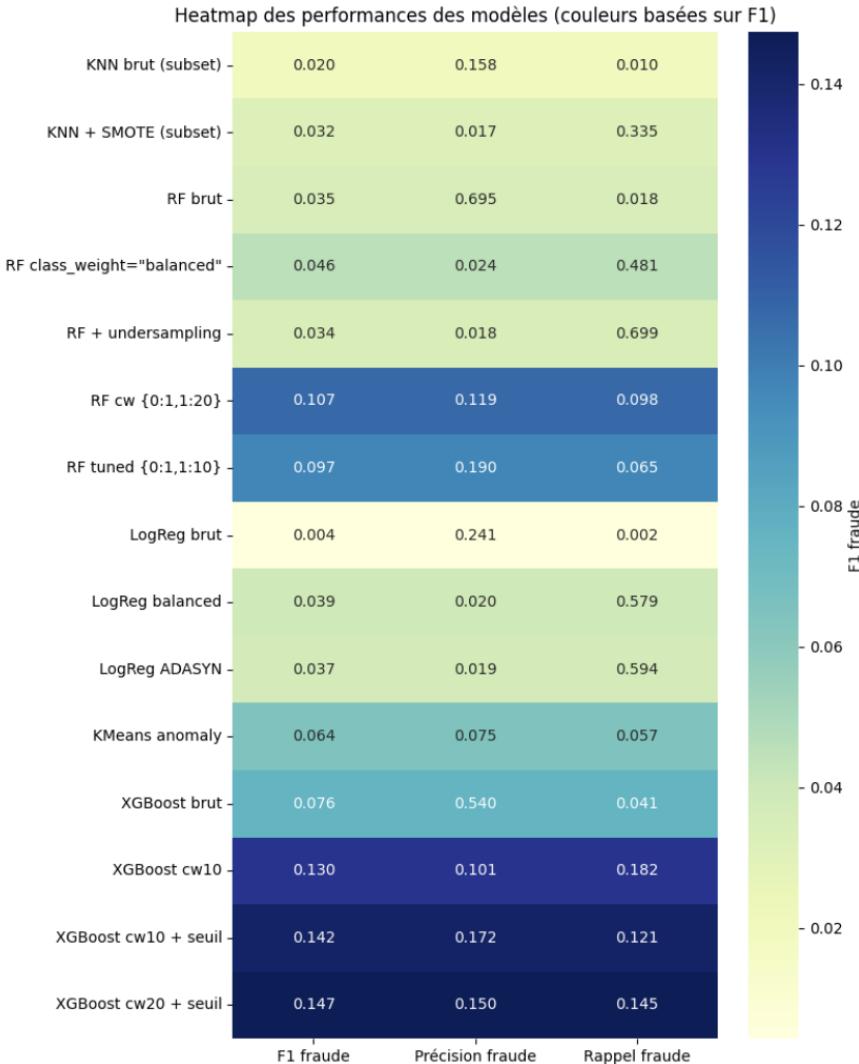


FIGURE 9 – Comparaison globale des modèles – F1-score sur la fraude

8.2 Analyse et interprétation

Les résultats montrent d'abord que les modèles bruts (sans gestion du déséquilibre) donnent une accuracy presque parfaite, mais un F1-score très faible sur la fraude : en pratique, ils laissent passer la majorité des fraudes et ne sont donc pas acceptables pour un service de conformité, même si les indicateurs globaux sont flatteurs. Les versions cost-sensitive, via `class_weight` ou `scale_pos_weight`, améliorent clairement le rappel fraude mais génèrent davantage de faux positifs ; côté métier, cela se traduit par plus d'alertes à traiter, avec un compromis à trouver entre charge opérationnelle et risque

de fraudes non détectées. Les approches avec sur-échantillonnage (SMOTE, ADASYN) poussent ce raisonnement à l'extrême : elles captent beaucoup plus de fraudes mais au prix d'une précision très faible, ce qui n'est viable que si l'organisation peut absorber un flux massif d'alertes injustifiées.

Les Random Forest cost-sensitive offrent un compromis intermédiaire : elles améliorent le F1 fraude par rapport à la RF brute, tout en restant relativement stables sur la classe non-fraude, mais restent globalement en retrait par rapport aux meilleures configurations XGBoost. Les modèles **XGBoost cost-sensitive avec seuil optimisé** fournissent le meilleur F1-score sur la fraude tout en conservant une accuracy élevée ; ils apparaissent donc comme les candidats les plus crédibles pour une mise en production, car ils augmentent significativement la détection de fraudes sans dégrader excessivement l'expérience client (trop de transactions bloquées à tort). Enfin, l'algorithme K-means anomaly affiche des performances modestes en F1 fraude, mais il illustre l'intérêt d'une brique non supervisée qui pourrait être utilisée pour pré-filtrer des comportements atypiques ou expliquer des patterns de risque, en complément des modèles supervisés principaux.

9 Conclusion et perspectives

9.1 Synthèse des résultats

Le travail portait sur la détection de fraudes bancaires dans un contexte de déséquilibre extrême entre classes, où la grande majorité des transactions sont légitimes. Dans un tel cadre, l'accuracy globale est peu informative et la priorité est donnée au F1-score de la classe fraude, ainsi qu'au compromis précision/rappel. Les modèles « bruts » (sans traitement du déséquilibre) – régression logistique, Random Forest, XGBoost – obtiennent tous une accuracy très élevée, mais un F1 fraude faible, ce qui confirme qu'ils restent insuffisants pour un usage métier. Les approches cost-sensitive et/ou combinées à du rééchantillonnage (ADASYN, undersampling) améliorent nettement la détection de la fraude, au prix d'une augmentation du nombre de faux positifs. Parmi l'ensemble des modèles testés, les **configurations XGBoost cost-sensitive avec optimisation du seuil de décision** obtiennent les meilleurs F1-scores sur la fraude, tout en conservant une accuracy élevée et un comportement satisfaisant sur la classe non-fraude ; elles apparaissent ainsi comme les candidates les plus sérieuses pour une mise en production.

9.2 Limites du travail

Ce travail présente plusieurs limites. L'optimisation des hyperparamètres a été réalisée dans un contexte de ressources de calcul restreintes : quelques recherches aléatoires (`RandomizedSearchCV`) n'ont donc été menées que sur des espaces de paramètres limités, et certaines configurations reposent principalement sur des réglages manuels guidés plutôt que sur une exploration exhaustive. Par ailleurs, la forte asymétrie entre classes complique l'analyse : les performances (en particulier le F1-score fraude) restent sensibles au choix des métriques, des poids de classes et des seuils de décision, ce qui impose d'interpréter les résultats comme des compromis raisonnables plutôt que comme des optima définitifs.

9.3 Perspectives

Plusieurs pistes d'amélioration peuvent être envisagées. Sur le plan algorithmique, il serait naturel d'explorer des variantes plus avancées de gradient boosting (LightGBM, CatBoost, variantes d'XGBoost) et de comparer leur comportement à celui du meilleur modèle actuel sur la fraude. Du côté des approches non supervisées, des méthodes d'anomaly detection plus sophistiquées – telles qu'Isolation Forest, des auto-encodeurs ou des méthodes hybrides combinant clustering et scoring – pourraient compléter ou renforcer la détection des comportements atypiques au-delà de K-means. Enfin, sur le volet optimisation, une recherche d'hyperparamètres plus systématique (Randomized search à plus grande échelle, voire optimisation bayésienne) permettrait d'affiner les meilleurs modèles, en particulier pour XGBoost et les forêts aléatoires, sous réserve de ressources de calcul suffisantes.

A Annexe : Meilleurs modèles par dataset

Dataset	Imb.	Best KNN	F_1 KNN	Best linéaire	F_1 lin.	Best non linéaire	F_1 non lin.
abalone17	True	knn_smote	0.071 ± 0.031	svm_lin_balanced	0.099 ± 0.011	gb_adasyn	0.097 ± 0.024
abalone20	True	knn_smote	0.068 ± 0.025	svm_lin_balanced	0.070 ± 0.007	rf_adasyn	0.115 ± 0.081
abalone8	True	knn_smote	0.324 ± 0.023	logreg_balanced	0.369 ± 0.013	gb_adasyn	0.385 ± 0.018
australian	False	knn_cv	0.819 ± 0.023	svm_lin_base	0.847 ± 0.007	rf_base	0.848 ± 0.023
autompq	False	knn_scaled	0.782 ± 0.052	logreg_l1	0.840 ± 0.027	gb_base	0.875 ± 0.046
balance	False	knn_cv	0.938 ± 0.034	logreg_base	0.949 ± 0.017	ada_base	0.982 ± 0.012
bankmarketing	True	knn_smote	0.464 ± 0.007	svm_lin_balanced	0.556 ± 0.010	gb_adasyn	0.567 ± 0.015
bupa	False	knn_base	0.560 ± 0.051	svm_lin_hinge	0.596 ± 0.044	ada_base	0.685 ± 0.032
german	False	knn_cv	0.433 ± 0.039	logreg_l1	0.591 ± 0.023	gb_base	0.573 ± 0.024
glass	False	knn_scaled	0.688 ± 0.032	svm_lin_base	0.621 ± 0.104	gb_base	0.785 ± 0.054
hayes	False	knn_cv	0.653 ± 0.274	logreg_l1	0.761 ± 0.070	ada_base	1.000 ± 0.000
heart	False	knn_cv	0.783 ± 0.031	svm_lin_base	0.791 ± 0.046	ada_base	0.780 ± 0.014
iono	False	knn_cv	0.782 ± 0.088	svm_lin_base	0.834 ± 0.041	rf_base	0.914 ± 0.020
libras	True	knn_cv	0.811 ± 0.093	svm_lin_hinge	0.863 ± 0.068	rf_adasyn	0.817 ± 0.088
newthyroid	False	knn_cv	0.917 ± 0.023	svm_lin_base	0.808 ± 0.064	rf_base	0.932 ± 0.024
pageblocks	True	knn_cv	0.837 ± 0.026	svm_lin_hinge	0.740 ± 0.029	rf_base	0.889 ± 0.011
pima	False	knn_scaled	0.582 ± 0.035	logreg_base	0.609 ± 0.015	rf_base	0.596 ± 0.040
satimage	True	knn_cv	0.696 ± 0.024	svm_lin_balanced	0.290 ± 0.009	rf_adasyn	0.692 ± 0.041
segmentation	True	knn_cv	0.898 ± 0.027	svm_lin_hinge	0.644 ± 0.030	rf_base	0.920 ± 0.009
sonar	False	knn_cv	0.818 ± 0.059	logreg_l1	0.775 ± 0.025	ada_base	0.827 ± 0.013
spambase	False	knn_scaled	0.881 ± 0.006	svm_lin_hinge	0.911 ± 0.007	rf_base	0.938 ± 0.005
splice	False	knn_cv	0.803 ± 0.016	logreg_l1	0.841 ± 0.010	gb_base	0.969 ± 0.005
vehicle	False	knn_cv	0.857 ± 0.028	logreg_base	0.928 ± 0.026	rf_base	0.934 ± 0.036
wdbc	False	knn_cv	0.947 ± 0.033	logreg_base	0.961 ± 0.019	ada_base	0.949 ± 0.023
wine	False	knn_cv	0.963 ± 0.023	svm_lin_base	0.994 ± 0.013	rf_base	0.983 ± 0.015
wine4	True	knn_smote	0.103 ± 0.041	logreg_balanced	0.133 ± 0.024	rf_adasyn	0.140 ± 0.056
yeast3	True	knn_cv	0.780 ± 0.033	svm_lin_hinge	0.764 ± 0.011	gb_base	0.795 ± 0.014
yeast6	True	knn_distance	0.565 ± 0.103	logreg_base	0.387 ± 0.177	ada_base	0.477 ± 0.160

TABLE 8 – Meilleur modèle par famille et par dataset (F_1 moyen ± écart-type).